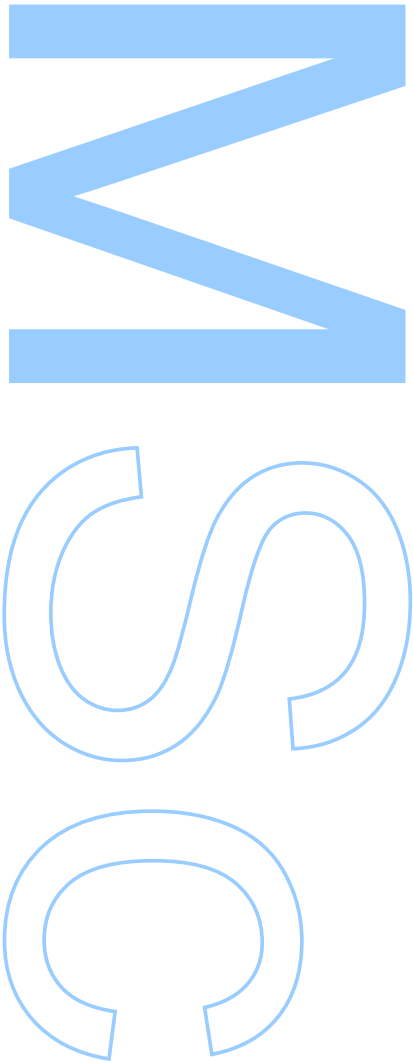
Secure Cloud Storage in Android Devices

Paulo Alexandre Carvalho Ribeiro
Master's Degree in Network and Information Systems Engineering
Departamento de Ciência de Computadores
2017

Orientador
Sérgio Crisóstomo, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

Coorientador
Rui Prior, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Dedicado aos meus pais, à minha namorada Filipa Crespo e ao meu avô Manuel Veloso de Carvalho.

Acknowledgments

I would first like to express my sincere gratitude to my supervisors, Prof. Rui Prior and Prof. Sérgio Crisóstomo, for all of their support, time and advice. They always allowed this paper to be my own work, but steered me in the right direction whenever I needed it.

I would also like to thank my friends for all the good moments during this years.

Finally, I must express my very profound gratitude to my parents and to my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Abstract

Digital data storage is essential nowadays. We store all types of data in our devices, either locally or using cloud storage services. Cloud services have several advantages, such as data sharing among devices, space saving in local storage, and data preservation in case of devices hardware failure. However, they also pose some risks, which users often do not realize, such as temporary or permanent unavailability or loss of confidentiality of the stored files. This work consists on the development of a secure file storage system based on public cloud services that mitigate the mentioned risks by combining the use of multiple cloud providers with redundancy mechanisms and cryptographic techniques. The system ensures that, even if one provider is hostile or goes out of business, there is no loss of data or confidentiality.

Resumo

O armazenamento de dados em formato digital é actualmente indispensável. Guardamos diversos tipos de dados nos nossos dispositivos, quer localmente, quer recorrendo a serviços de armazenamento cloud. Os serviços cloud apresentam várias vantagens, nomeadamente a partilha de dados entre dispositivos, a poupança de espaço no armazenamento local, e a sobrevivência dos dados a falhas de hardware dos dispositivos. No entanto, o seu uso apresenta alguns riscos, dos quais muitas vezes os utilizadores não se apercebem, sendo os mais significativos os de indisponibilidade temporária ou permanente dos ficheiros e de perda de confidencialidade dos mesmos. Este trabalho consiste no desenvolvimento de um sistema de armazenamento seguro de ficheiros em serviços públicos de cloud que mitiga os riscos mencionados combinando o recurso a vários provedores de cloud com o uso de mecanismos de redundância e de técnicas criptográficas. O sistema garante que, mesmo se um dos provedores for hostil, não há perda dados nem da sua confidencialidade.

Contents

List of Tables

List of Figures

Listings

Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

CBC Cipher Block Chaining

CSP Cloud Storage Provider

CSS Cloud Storage Service

FFT Fast Fourier Transform

FS File System

FUSE File System In User Space

HMAC Hash-based Message Authentication Code

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IDA Information Dispersal Algorithm

IV Initialization Vector

LRU Least Recent Used

MAC Message Authentication Code

NIST National Institute of Standards and Technology of U.S.

OS Operative System

PaaS Platform as a Service

SaaS Software as a Service

SELinux Security-Enhanced Linux

SHA Secure Hash Algorithm

Chapter 1

Introduction

Data storage is essential nowadays. We store everything, from personal data (like photos or text messages) to critical data (like medical records or financial reports). There are two main approaches to store data: local storage or cloud storage.

The concept of storing data in the cloud emerged in the last past few years, embedded in the cloud computing concept. Cloud computing is defined by National Institute of Standards and Technology of U.S. (NIST) as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [1].

Cloud storage is a particular case of cloud computing where the computing resources are storage servers. In this context, we can define a Cloud Storage Provider (CSP) as the entity that provides the service, a Cloud Storage Service (CSS) as the service that enables to store and manage the remote files, and cloud as the infrastructure where we store the data.

Many CSSs are available for use, either directed at users or companies. For users, there are many free options with a decent amount of space (at least 15GB), and for companies, as they need a lot more of space and speed, there are a lot of paid services that usually charge for bandwidth and storage capacity.

Storing data in the cloud has some advantages: we can overcome the limitation of the local

device storage capacity: the files are readily available from anywhere in any device and the cost is lower, since maintaining a local data center is expensive. Despite all the advantages, there are some drawbacks in storing data in the cloud. Local storage is more secure [2] [3], because when we store data in the cloud, it is stored in a place we do not know and supervised by people we do not know. Therefore, we can not be sure that they (the CSP) will not lose it, misuse it, or even leak it.

Some recent accidents show us that we can not trust in the CSPs to keep our data safe. Sometimes they fail or are hacked, and that can result in a leak, temporary or even permanent loss of stored data. In the 2016, every major commercial CSPs was offline for some time as we can see in Figure 1.1. The Google Cloud Platform was offline for more than 11 hours [4] [5]. In 2009 a company lost 500GB of users data along with the backup and never managed to recover the data [6]. In 2012 two-thirds (68 million) of Dropbox users passwords were stolen by hackers and later in 2016 they were made publicly available on the Internet [7]. In 2014 one of the most mediatic hacks ever occurred: hundreds of private photos of celebrities were leaked from the Apple Icloud storage service [8].

Another big concern about keeping data in the cloud is the vendor-lock-in issue. There is a possibility that the CSPs raise the price or simply start charging for bandwidth (in the case of the free ones), so if we have the data stored at only one provider, we either lose the access to the data or we are forced to pay a large amount of money to recover it. For more detail on that subject read reference [9]. To conclude, we do not have guarantees of confidentiality or availability of the data stored in public clouds.

This work consists of the development of an Android application that, using information partition and cryptographic techniques, allows users to keep folders in CSSs guaranteeing the confidentiality, availability, and integrity of its contents. Caching techniques are used to improve application performance. The objective is to use public cloud services like Dropbox, Google Drive and One Drive as storing platforms (public CSSs), although in this we work we present a proof of concept using diverse accounts of the same CSP.

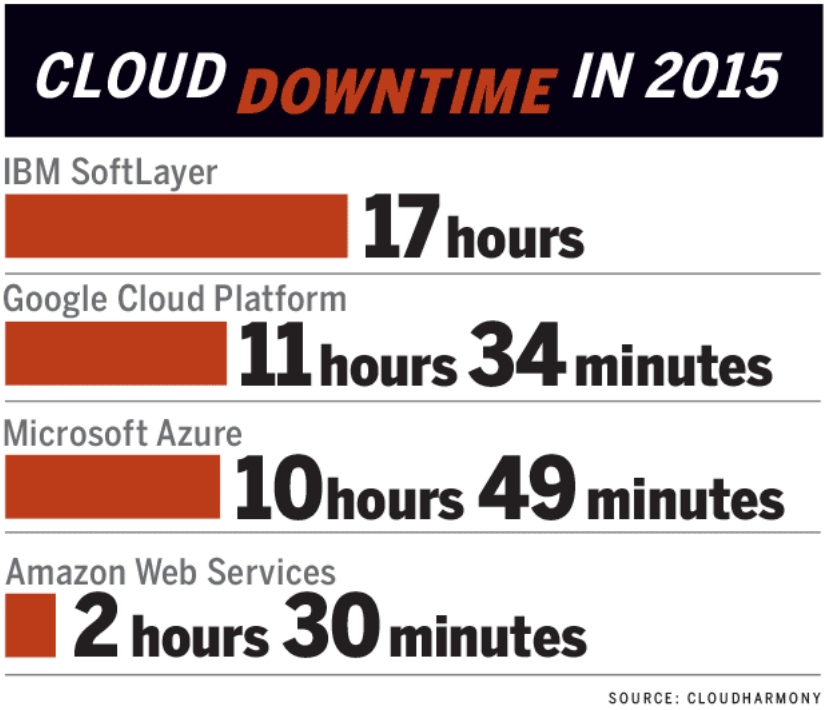


Figure 1.1: Cloud storage providers downtime in 2015. [4]

1.1 Context and Motivation

In the past years, we have witnessed a big growth of smartphones usage, as shown in Figure 1.2. We already reached a point where we have more smartphone users than desktop users [10]. We started to do a little bit of everything in our smartphones, and because of that we need to store a lot of data in them, data that sometimes is sensitive or merely private, and we want it to stay confidential. Either because we need to access them in other devices or because our device has limited storage space, we often store data in the cloud. Because of that, cloud storage is widely used in smartphones, but not the way we intend to use it, which is both explicit, where the user knows that folder is a "cloud" folder, but we want the use of that folder to be transparent (i.e., similar to anything else), and generic (any files and applications). That means that users are storing data in those services, most of them without knowing the risks involved.

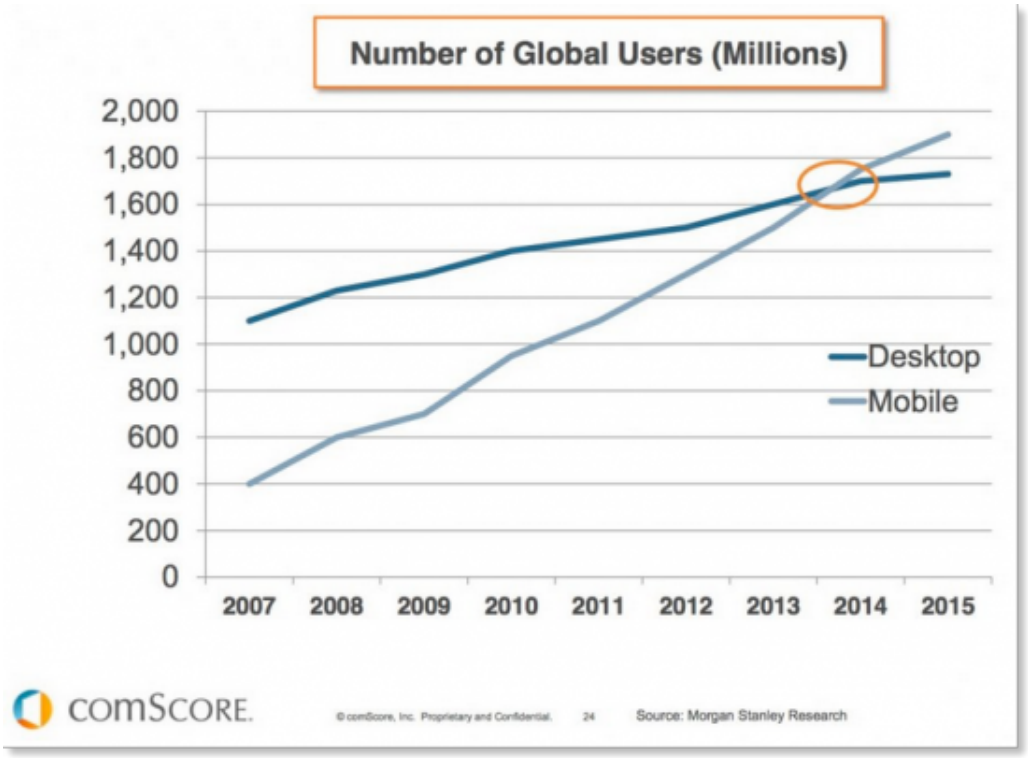


Figure 1.2: Mobile users vs Desktop users. [10]

We developed a solution for Android, that is the most used Operative System (OS) for smartphones with an 87.6% market share [11]. It is free to develop for Android unlike some other platforms, and it is one of the easiest mobile platforms to develop for.

Our system allows users to substantially reduce the risks of data loss, data leakage, and temporarily data unavailability. Data privacy is a big concern for most of the mobile device users, and they have no available solutions to address this problem. With our application, they can overcome the limitations of simple cloud storage, and securely store their data in the cloud.

1.2 Objectives

Our main goal is to develop a secure cloud based file system, where secure means assured confidentiality and availability. From that objective, we derive other objectives and requirements for the application.

We can categorize these requirements by their importance level:

Must Have

Confidentiality The data stored in the public clouds must stay confidential, even in a case of a partial leak (for example one cloud authentication compromised);

Availability The data stored in the public clouds must stay available, even if one of the CSPs is not. Using a single CSP the files stay unavailable in average 0.01% of the time, we aim to lower that unavailable time in our system;

Integrity The application must ensure data integrity. In the event of data manipulation (by the CSP or another agent) the application must be able first to detect then recover from that situation without data loss;

POSIX file system The application must be capable of mounting a POSIX [12] compatible file system that acts as a local folder, but in reality store the data in at least two public clouds;

Caching The system must maintain a local cache, so it does not need to download every file every time we need to access them.

Should Have

Good Performance The file system should have a decent performance in computational context; We must analyze the options that we can use to perform the operations to add redundancy to the system and to encrypt the files and take in consideration their performance, in order to minimize the impact of the user experience;

Low Storage Overhead To achieve data availability we use redundancy mechanisms that always have storage overhead associated, and as we are using cloud services to store the data (with limited free space), we need to pick a redundancy mechanism with an storage utilization rate near to optimal. Where the $utilization\ rate = \frac{usable\ storage}{total\ storage}$, the usable storage is the amount of space available to store files $(usable\ storage = total\ storage - redundancy\ storage)$.

Easy to use Interface The application must have a graphical interface that allows the users to configure and manage the system, allowing them to mount the File System (FS) in a local folder.

Could Have

Recover to fully consistent state In case of a CSP permanent failure, the system must be capable of automatically recover to a fully consistent state, asking the user for a new CSP only.

1.3 Dissertation Structure

In Chapter 2 we introduce some important definitions and concepts about cloud storage as they are a must to better understand the rest of the document. We also analyze the available options to achieve the main objectives (cloud storage integrity, availability and integrity), comparing the pros and cons of each option. Finally, we take a look into some implementations that somehow are related to the work described in this document, highlighting the similarities and differences.

In Chapter 3 we make a top down description of the system architecture, explaining the main components of the system and the chosen methods to achieve cloud storage integrity, availability and integrity.

In Chapter 4 we describe the development of the system, showing some code and explaining all the functions we implemented to achieve cloud storage confidentiality, availability and integrity.

In Chapter 5 we analyze the results of this work, highlighting the system performance and the impact of the decisions we made in the security and performance of the system.

In Chapter 6 we make a conclusion of the dissertation, discussing some important decisions and how we met the objectives. We also present the future work that can be made in order to improve the system.

Chapter 2

Background

In this chapter, we describe the mechanisms that can be used to achieve the proposed goals, by answering "What can be done to achieve confidentiality, availability, and integrity in a cloud of public clouds file system?". As well in this chapter are introduced some basic terms that will be used throughout the dissertation. Finally, we describe the current implementations and research already done on this subject.

2.1 Cloud Computing

As our primary goal is to build a secure cloud storage file system, we need to understand what is cloud storage, and for that, it is necessary first to understand cloud computing. Cloud storage is a particular service among all cloud computing possibilities.

We already introduced a formal definition of cloud computing in Chapter 1. In short, we can define it as an agglomeration of computing resources (usually servers), that can be deployed in a short period. The users of a cloud computing service can start by buying a small amount of those resources and, as their business grows, buy more and have access to them almost instantly with or without minimal service provider intervention. So they have access to an easily scalable service, usually cheaper than a local infrastructure.

2.1.1 Essential Characteristics

Cloud computing services must have some essential characteristics as [1]:

On-demand service A consumer can automatically provision computing resources (such as server time, network storage or virtual memory) without requiring human interaction on server side;

Broad network access The service is available over the network (internet) to be accessed by standard client platforms like mobile phones, tablets, laptops, and workstations;

Resource pooling The provider computing resources (such as storage, processing, memory, and network bandwidth) are pooled to serve multiple consumers, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. Users usually have no control or knowledge over the exact location of the provided resources, but may be able to specify location at a higher level of abstraction (country, state, or datacenter);

Rapid elasticity In order to enable consumers to be able to scale their platform quickly, the computing resources are elastically provisioned and released, in some cases automatically. The resources available for provisioning often appear to be unlimited in consumer view and can be appropriated in any quantity at any time.

2.1.2 Service models

We can classify the cloud services in three different ways [1] [13], depending on what we have access to:

SaaS The consumer can only use a provider application that runs on the cloud infrastructure. The application is accessible from an interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities;

PaaS The consumer can deploy applications created using programming languages, libraries, services, and tools supported by the provider to the cloud infrastructure. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment;

IaaS The consumer can provision processing, storage, networks, and other fundamental computing resources and can deploy and run arbitrary software in that resources, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

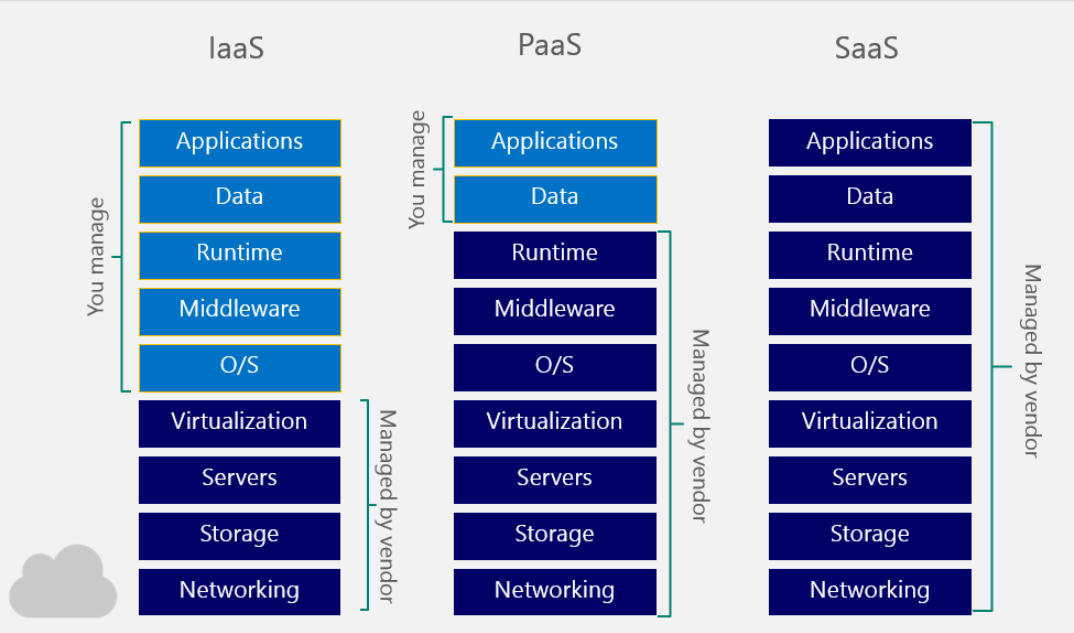


Figure 2.1: A comparsion between the discribed cloud service storage models SaaS, PaaS and IaaS [14], where vendor is the CSP.

In figure 2.1 we have a visual description of the presented models. As shown, each service model is very different. We can have cloud services that only allow us to use an application, but also can have services that allow us to control almost every aspect of the system. In our work, we use CSS's, and they are in the SaaS category, so this immediately imposes some restrictions to what we can do. We must be aware that we can not implement new functions as we are restricted to

the interface of the service provider.

2.1.3 Cloud Storage

Cloud storage is a cloud computing service, where the consumers can allocate an apparently infinite amount of storage space. The service is accessible from an interface, such as a web browser or application. Some CSPs also have a Application Programming Interface (API) (usually a REST API) that allow consumers to manage the contents stored in the cloud. As we can see in Figure 2.2, the CSP can accept objects with different granularity: they can be blocks or files depending on the CSP.

The consumer does not manage or control the underlying cloud infrastructure, and he only can allocate space and use it to store data, so cloud storage services are labeled as SaaS.

There are a lot of CSSs, some targeting end users while others targeting companies, usually the ones that target the users give some free space to consumers, then if they need to pay for extra space.

Our service is based on CSPs that target end-users, as DropBox, Google Drive and One Drive, which are free to use public clouds.

We can look at cloud as a folder where we can store files and other folders, just as a folder in a local file system. We have API calls that allow us to get, put, delete and move files stored in the cloud, as we can do in a local file system.

2.2 Storage Availability

In order to mitigate the problem of availability we can add redundant information to the file system, so when a CSP fails (temporarily or permanently), we can still access the files. In this section, we look into some different methods to achieve data redundancy, each one with different characteristics. The probability that more than one cloud service provider fail at the same time is low, considering that the average downtime of public CSPs is 0.001% [4], the probability of having two unavailable at the same time is 0.00001%. Considering that, ensuring that our system can recover from one failure should be enough, although we analyze some options that can tolerate

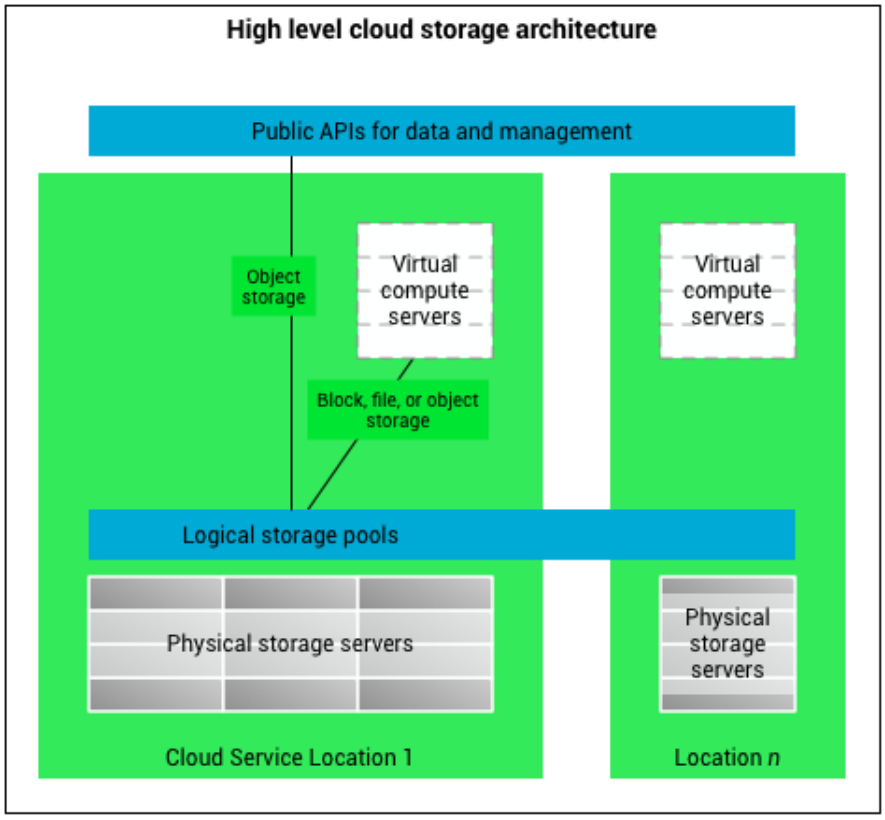


Figure 2.2: A high level architecture of a cloud storage. [15]

more than one failure.

As we use more than one CSP, we can split the files among them, and this adds a confidentiality factor because if anyone can access our files, they do not have access to the entire content. Some analyzed methods inclusively add some obfuscation to the file chunks, making it even harder to get some information of the original content. But the confidentiality factor provided by these methods is not much important because it is still possible to recover some information of the original data from individual chunks, for achieving confidentiality we present other options in section 2.3.

2.2.1 Mirroring

The most straightforward approach to obtain redundancy in a file system is to make copies of the original content. That method is known by information replication or mirroring. We can make one or more copies of the files, and put them in the available cloud storage providers and when

one provider fails we just get the file from any other. This is the technique used by RAID 1 [16] and it requires at least two disks. We can apply this method to our system considering individual clouds as disks.

Replication is one of the most common methods to achieve high data availability. For example, Google File System [17] and Hadoop distributed file system [18] use replication to achieve data availability. It is easy to implement and has excellent performance. Also, it is a very flexible solution when it comes to the number of failures that we can tolerate. If we have n ($n >= 2$) clouds we can tolerate up to $n - 1$ failures at same time. But this comes with a huge drawback, and it has a high storage overhead. For example, with the original data and 3 replicas, the storage utilization rate is only $\frac{1}{1+3} = 25\%$. Another drawback associated with the storage overhead is the network bandwidth required to send the replicas to the respective clouds, or to migrate all the data.

2.2.2 Simple Parity

Simple Parity is a straightforward approach to achieve one drive fault tolerance and has been used for a long time by RAID 4. As shown in Figure 2.3 it is based on simple parity checks, where one disk is reserved for parity storage. Considering that we have three drives and one is reserved for parity storage, to store a file, we first split the file into two chunks, calculate the parity between them (originating a new piece of the same size), doing bit by bit XOR operations [19].

This method requires at least three clouds, if one CSP fails we can rebuild the information that was stored in it by doing a XOR operation between the remaining two disks.

The storage overhead with this method is optimal, the total amount of usable space is equal to the sum of the capacity of all clouds minus the storage of the parity reserved cloud. Considering three clouds with 1 GB each, we have 2GB of usable space in a total of 3GB, the storage utilization rate is $\frac{2}{2+1} = 66\%$. It is an easy to implement method, based in simple operations with a good performance.

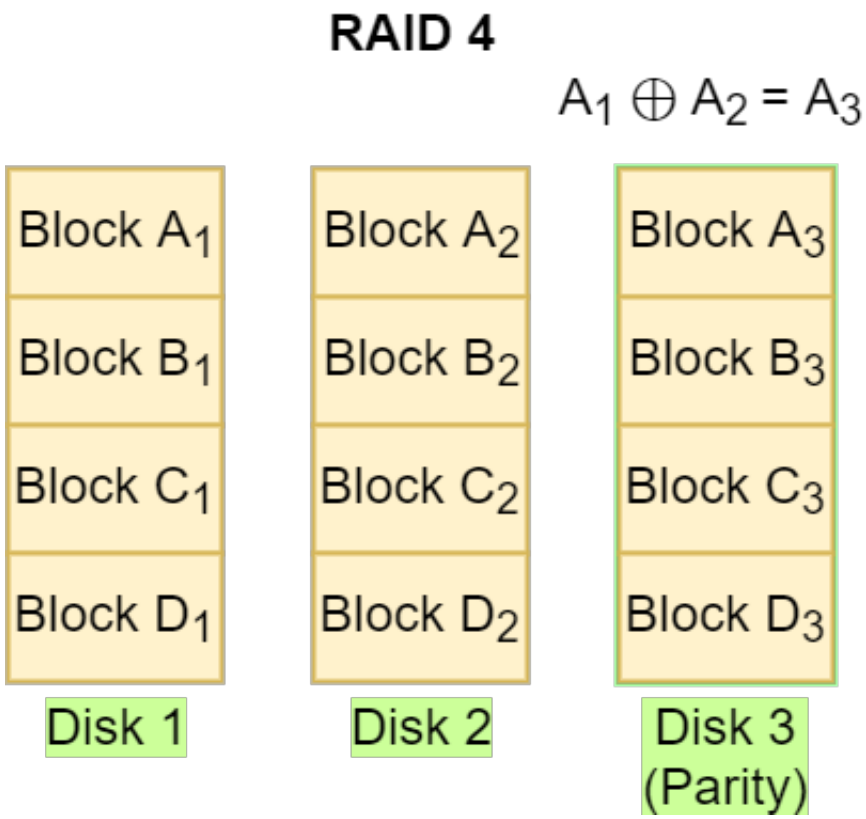


Figure 2.3: An example how parity works in RAID 4 scheme with 4 disks where one is reserved for parity storage. [20]

2.2.3 Erasure Coding

Erasure coding is a coding technique for encoding a file adding redundant information to it. It allows us to split a file into k chunks and encoded it into $n > k$ chunks, where, considering $|F|$ the size of the file, each chunk has a size of $\frac{n}{k} \times |F|$, that is an optimal storage overhead if we are recovering the file from k chunks. Then from any k chunks out of the n chunks, we can rebuild the original file as we can see in Figure 2.4. We denote $m = n - k$ as the number of redundant or parity chunks and assume that each chunk is stored on a distinct cloud. In most implementations of this method we can chose any combination of (n, k), where $n > k > 1$.

As there is an encoding process, the originated chunks are not plain text, making it harder to get information about the original data.

Considering that we have three clouds with 1GB each, we have 2GB of usable space in a total of 3GB, the storage utilization rate using this scheme $(n = 3, k = 2)$ is $\frac{2}{2+1} = 66\%$.

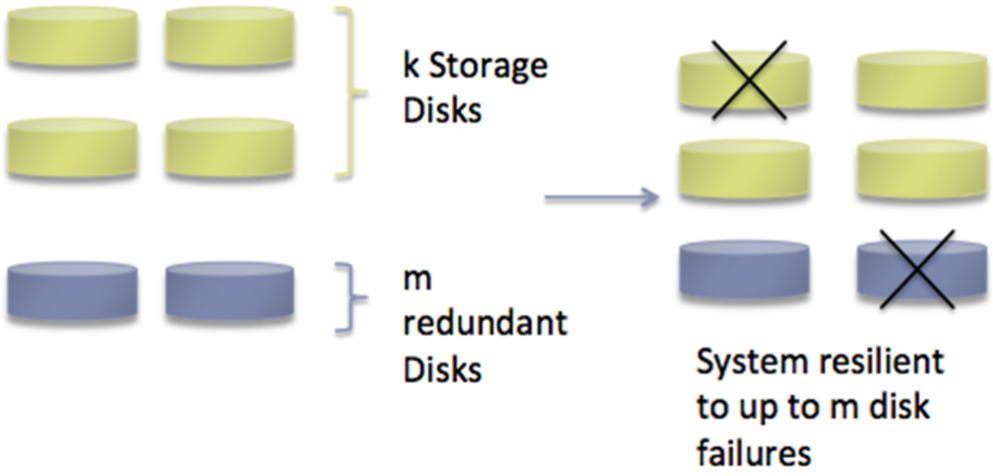


Figure 2.4: Example of a system with an erasure code where k=4 and n=6, resulting in a system that is resilient to m=2 failures. [21]

This is the generic idea of erasure coding, and there are many implementations out there, the algorithms that implement this technique are known as IDAs [22]. The difference between the implementations is essentially the encoding algorithm, which can result in different performances.

Those algorithms typically are computationally intensive, many of them are formulated as the matrix-product forms, which means that the encoding and decoding complexities depend on the matrix products computing overhead. The conventional approaches of (n, k) IDA, such as [23], requires $O(n \times k)$ operations for encoding and $O(k^2)$ operations for decoding. But there are some recent implementations with much better performance, for example, there is an implementation based in Fast Fourier Transform (FFT) over finite fields, with a computational complexity of $O(n \log n)$ for encoding and $O(k(n - k + \log k))$ for decoding [24]. Another one is based in Reed-Solomon erasure coding [25], a very popular erasure coding technique, it has a computational complexity of $O(n \log n)$ for encoding and decoding [26].

Although the storage overhead is optimal, and there are some implementations with proper encoding and decoding performance, they are complicated to implement.

2.3 Storage Confidentiality

The access to files in a cloud is already protected by a security mechanism (Username and Password), and some CSPs claim that they encrypt the files, but we do not know who has access to the cryptographic keys. Also, the authentication mechanism is not enough to ensure the confidentiality of the files, and there are many examples where this mechanism failed, already described in section 1.1.

Throughout this dissertation, we will consider that the attacker has not access to the user device, we aim to protect the user from inside attacks (from the CSPs) and from attacks to the CSPs. With that in mind we use the user device to store some critical information as CSP account authentication tokens and cryptographic keys.

We also have seen in section 2.2 that the mechanisms of redundancy can add some security, but we want a robust system, where in a case of a leak, of the entire content of a single cloud, it does not result in a leak of any information about the original files. So we assume the worst case scenario, where the cloud storage provider can be malicious (inside attack) and/or the authentication details can be leaked or compromised.

In order to protect the files stored in the cloud from inside security breaches and external attackers we need to use cryptography techniques, but we need to take some important aspects in mind when choosing a cryptographic method:

Performance The chosen cipher must be fast, every time we want to put, modify or retrieve a file from a cloud storage service. We will need to perform cryptographic operations, and there will always be a delay associated with those operations, so we need to choose one cipher with a short time overhead. There is as well a computational overhead associated with those operations and considering that the system will run on mobile devices, we need to take this aspect into account, the computational power is much more limited than in other devices, and they run on a battery;

Storage Overhead The chosen cipher must add a minimal space overhead, although the cloud storage providers that we use are free they have limited space, so we need to take this aspect in mind.

Modern cryptography relies on a secret called key, that is the security factor needed to encrypt and decrypt the files. But there are two different approaches, the symmetric ciphers, that use only one key that must be known by the sender and the receiver, and public key cryptography that use a pair of keys (one private and one public).

2.3.1 Symmetric Cryptography

Symmetric ciphers require only one key, and that key must be known by the sender and the receiver, as the encryption and decryption process relies on that key, if the key is revealed, the data can be revealed too.

There are two types of symmetric ciphers that are classified by the granularity of the encrypt and decrypt process:

Stream ciphers The bytes are individually encrypted without feedback to other chunks of data (in most ciphers/modes). Also, stream ciphers do not provide integrity protection or authentication, whereas some block ciphers can provide integrity protection, in addition to confidentiality. This method is used to encrypt stream of data with no predefined size;

Block ciphers The encryption unit is a block with a pre-defined size, most of the implementations have a feedback system where the result of a block influences the next one, they are more susceptible to noise in transmission, that is if you mess up one part of the data, all the rest is probably unrecoverable.

Block ciphers typically require more memory, since they work on larger chunks of data and often have "carry over" from previous blocks, whereas since stream ciphers work on only a few bits at a time, they have relatively low memory requirements (and therefore cheaper to implement in limited scenarios such as embedded devices).

Because of all the above, stream ciphers are usually best for cases where the amount of data is either unknown or continuous - such as network streams. Block ciphers, on the other hand, are more useful when the amount of data is known before the transmission, as a file, or request/response protocols, such as Hypertext Transfer Protocol (HTTP) message where the length of the total message is known.

2.3.2 Public Key Cryptography

Public key cryptography requires a pair of keys, where one is to be kept private and one to be publicly known. This kind of ciphers allows us to have different keys in the encryption decryption process.

To send a file to someone, we use their public key to encrypt it then only the person that has the private key that corresponds to that public key can decrypt the file.

Those ciphers are great because we can exchange a secret without a previously known key. We just need a protocol to distribute the public keys among the users. However, it is not suitable for large amounts of data because of its poor performance when compared to symmetric ciphers [27].

2.4 Storage Confidentiality and Availability Combined Solutions

In this section, we look into a solution that provides at same time confidentiality and redundancy known by secret sharing scheme. We describe as well how we can combine the previously discussed techniques in sections 2.2 and 2.3 to achieve storage confidentiality and availability. We will discard combinations that use public key cryptography because we already concluded that they are not adequate to a system like ours.

Finally, we make a resumed comparison between the possible solutions, analyzing the most important factors.

In some presented solutions we need to store, in the local storage, a key for a symmetric cryptographic algorithm. But this does not add much of a security risk because the system already need to store CSPs authentication information.

2.4.1 Secret Sharing

This scheme has very similar functional characteristics with erasure coding presented in Section 2.2. It also allows us to split a file into k chunks and encoded it into n chunks, then from any k chunks out of the n chunks we can rebuild the original file, but for secret sharing k must be $n = 2k - 1$ [28] instead of just $n > k > 1$.

The major difference between secret sharing and erasure coding is that with secret sharing having access to $k-1$ chunks of the file does not provide any information about the original file. So that allows us to use this method to achieve confidentiality and availability at the same time.

Although this approach has one big drawback when compared to erasure codes, the storage overhead, with secret sharing each chunk has the same size of the original file. Which regarding storage overhead is equal to the information replication approach that is bad.

2.4.2 Secret Sharing with Symmetric Cipher and Information Dispersal Algorithm

To solve the storage overhead problem of secret sharing, it was proposed a scheme [29] that combines traditional secret sharing schemes like [28] encryption and Information Dispersal Algorithms (IDA).

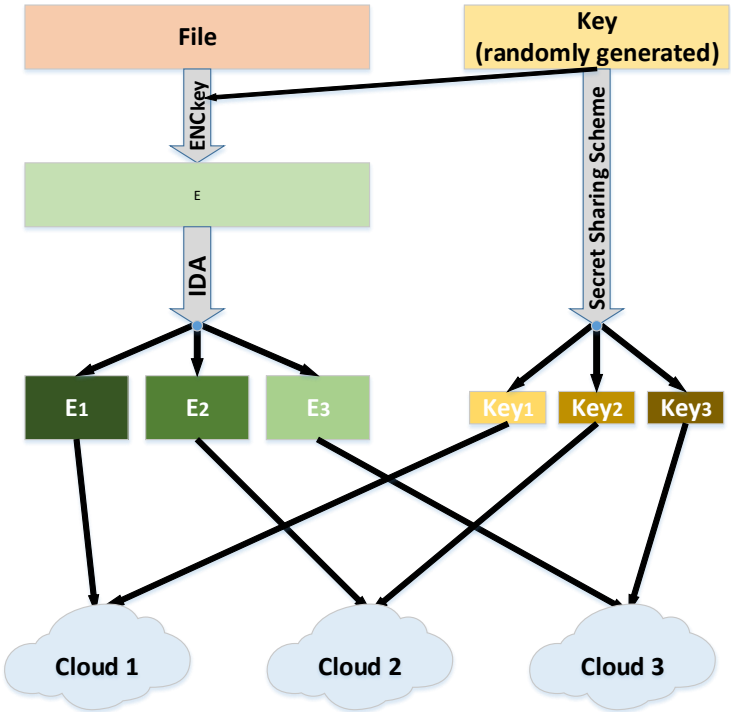


Figure 2.5: Process of storing a file in the cloud of clouds using secret sharing, IDA and a symmetric cipher, where $n = 3$, $k = 2$ and $m = 1$. ENC_{key} represents the encryption of the File using key.

With this combination we have a scheme with a minimal storage overhead, assuming $|F|$ as the size of a file F, each chunk has a size of $\frac{|F|}{m}$ where m is the number of chunks needed to recover

the file, plus a short piece of information that depended on the security factor (the size of the key of the cipher).

The process is simple and is illustrated in Figure 2.5. To assure confidentiality we generate a random key for the symmetric cipher algorithm, and encrypt the file using that key, then we use a IDA to encode the file that has a much lower storage overhead than a secret sharing scheme, and use the secret sharing scheme (with same k, n and m parameters) to encode only the key (that is much smaller when compared to the file). As a result, we have n chunks of the key encrypted with a secret sharing scheme and n chunks of the file encoded with a IDA. Now we can send them in pairs $(key_i, file_i)$ to the clouds, then to recover a file we just need k of those chunks to retrieve the key, recover the file, and then decrypt the file as shown in Figure 2.6.

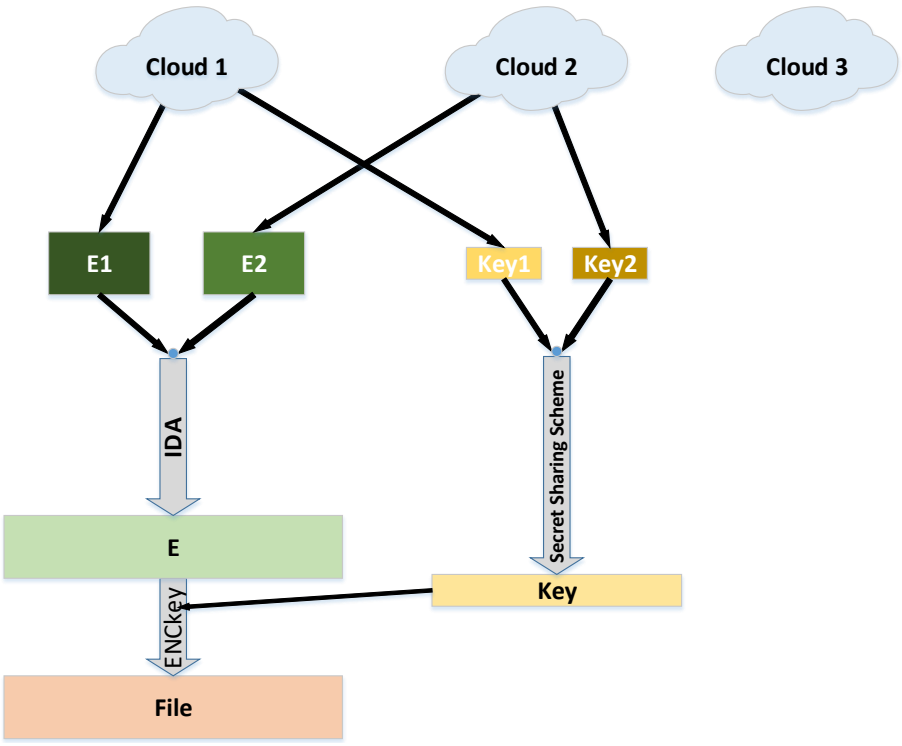


Figure 2.6: Process of recovering a file from the cloud of clouds using secret sharing, IDA and a symmetric cipher, where $n = 3$, $k = 2$ and $m = 1$. ENC_{key} represents the decryption of E using key.

With this scheme, in an eventual failure of one to m CSP we still have access to the files, only need to get other redundant chunks from other clouds. If $k - 1$ pairs $(key_i, file_i)$ are leaked there is not any given information at all about the original files [29].

2.4.3 Information Replication with Symmetric Cipher

This is the simplest scheme presented, the process to store a file is described in Figure 2.7, we use a symmetric cipher to encrypt the file with a key stored locally in the android device, then we just send one copy of the encrypted file to each cloud.

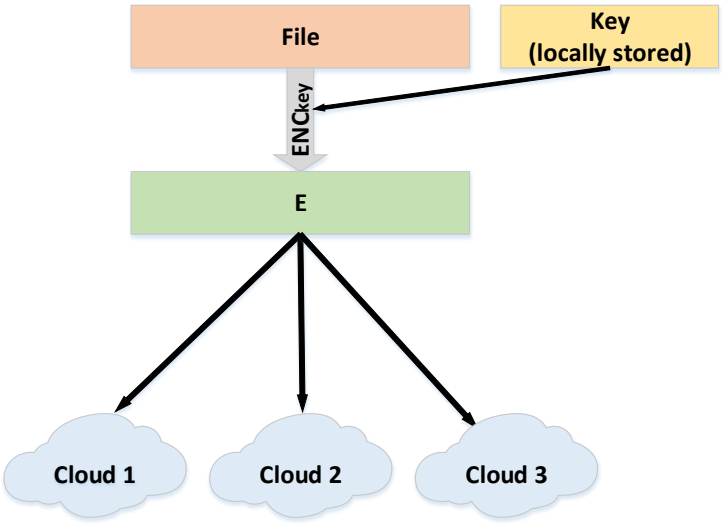


Figure 2.7: Process of storing a file in the cloud of clouds using information replication and a symmetric cipher, with 2 copies of the original data. ENC_{key} represents the encryption of E using key

To recover the file we just get a copy from any cloud as shown in Figure 2.8, if one fails we just get it from any other.

This solution tolerates up to $n-1$ cloud failures, where n is the total number of clouds. And the storage overhead is the same as the information replication without encrypting the file which is bad.

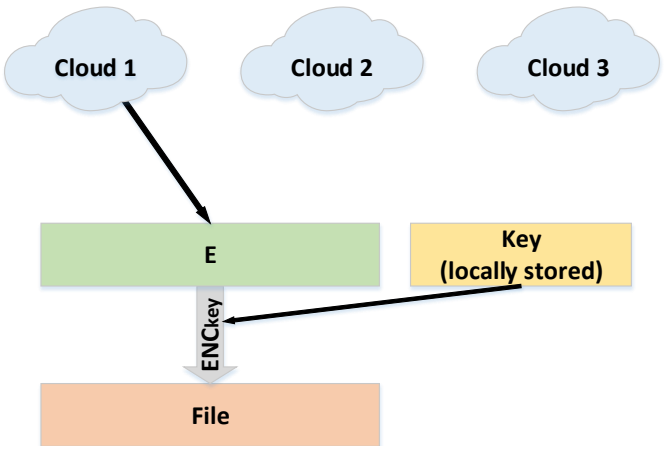


Figure 2.8: Process of recovering a file from the cloud of clouds using information replication and a symmetric cipher, with 2 copies of the original data. ENC_{key} represents the decryption of E using key.

2.4.4 Simple Parity with Symmetric Cipher

With this solution, we first encrypt the file (using a symmetric cipher) with a locally stored key, then split it into $n-1$ chunks, where n is the total number of available clouds, and calculate the parity of those chunks (with XOR operations) to get a redundant chunk. This process is described in Figure 2.9.

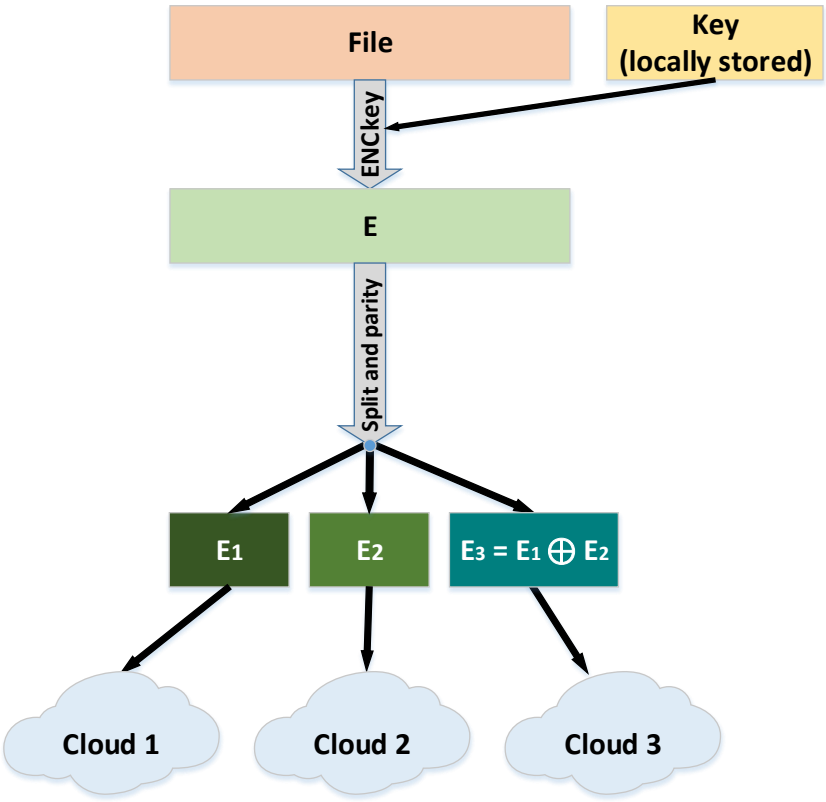


Figure 2.9: Process of storing a file in the cloud of clouds using simple parity and a symmetric cipher, with a total of three clouds. ENC_{key} represents the encryption of E using key, and cloud 3 stores the parity.

That redundant chunk allows us to recover from a single CSP failure, in that eventuality we need to calculate the parity between all the other chunks (including the parity one).

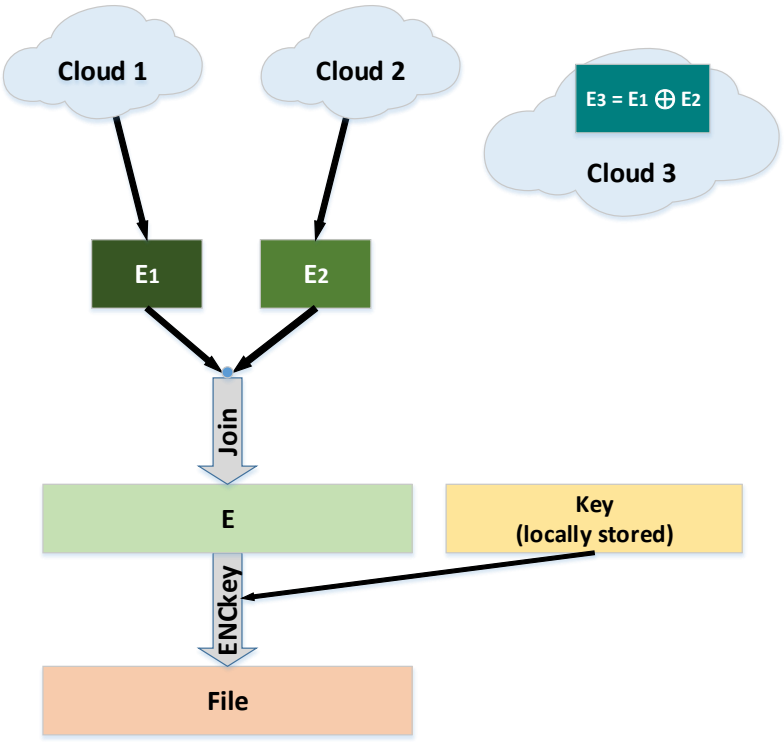


Figure 2.10: Process of recovering a file from the cloud of clouds simple parity and a symmetric cipher, with a total of three clouds. ENC_{key} represents the decryption of E using key, and cloud 3 stores the parity.

As we can see in Figure 2.10, we always need $n-1$ chunks to recover the file and do not need to do any calculations to recover it when all clouds are available (only need to join the chunks and then decipher). This allows us to have an excellent performance, but we are limited to a one failure resistance.

In the eventual failure of one CSP we must get the redundant chunk and recalculate the missing one as described in Figure 2.11;

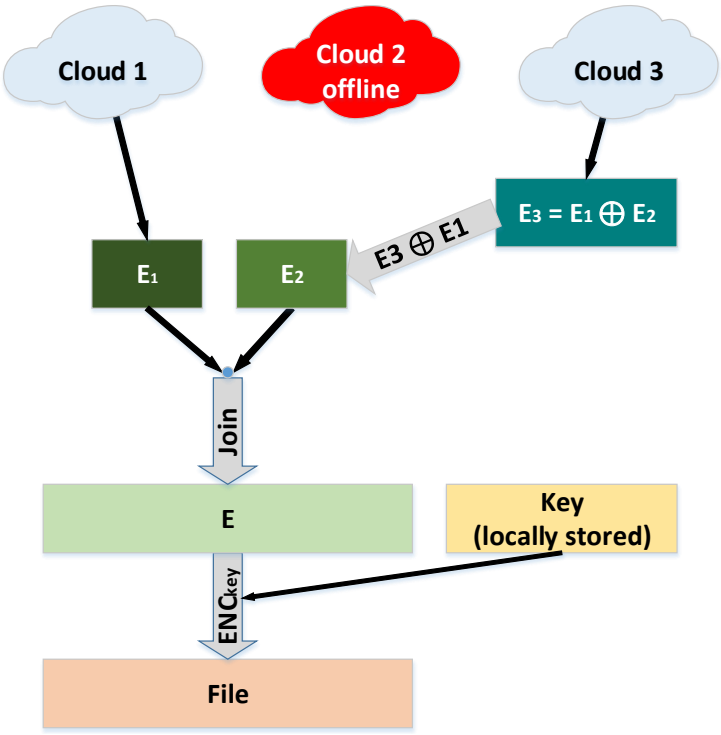


Figure 2.11: Process of recovering a file from the cloud of clouds considering the failure of one cloud, using simple parity and a symmetric cipher, with a total of three clouds. ENC_{key} represents the decryption of E using key, cloud 3 stores the parity chunks.

2.4.5 IDA with Symmetric Cipher

This method is somehow similar to the one presented in subsection 2.4.2, but instead of using a secret sharing scheme to ensure confidentiality distributing the key among the clouds, it stores the key of the symmetric cipher locally in the android device storage.

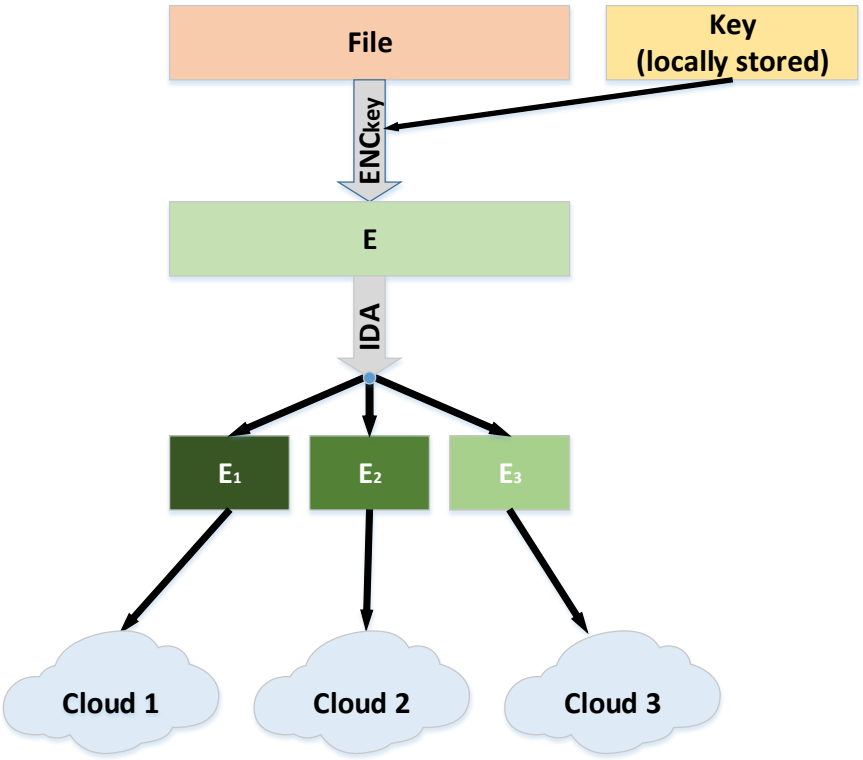


Figure 2.12: Process of storing a file in the cloud of clouds using IDA and a symmetric cipher, where $n = 3$, $k = 2$ and $m = 1$. ENC_{key} represents the encryption of the File using key.

We need first to define a key, then to store a file we encrypt it with that key, then encode it with a IDA. As a result, we have n chunks, where n must be equal to the number of clouds. Finally, we send each chunk to each cloud, the whole process is described in Figure 2.12. To recover the file, we get any k chunks from any k clouds, decode the chunks with the IDA, then decrypt the file as described in Figure 2.13.

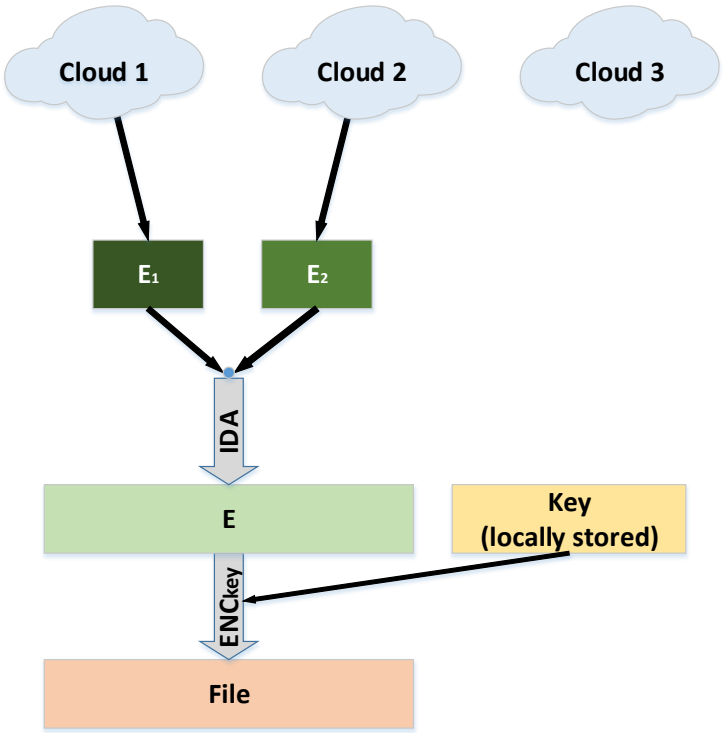


Figure 2.13: Process of recovering a file in the cloud of clouds using IDA and a symmetric cipher, where $n = 3$, $k = 2$ and $m = 1$. ENC_{key} represents the encryption of File using key.

As in the method described in Subsection 2.4.2, in an eventual failure of one up to m CSP we still have access to the files, and we only need to get other redundant chunks from the other clouds.

2.4.6 Solutions Comparison

In this subsection we objectively compare the differences between the presented solutions, considering important parameters that can help us understand their adequacy to meet our objectives.

For storage efficiency calculations we will assume $n = 3$ number of available clouds, k the number of chunks needed to recover the data, $m = n - k$ the number of redundant chunks.

In Table 2.1 we have a comparison between the presented solutions, the Implementation complexity column is and subjective indicative of the difficulty level expected to efficiently implement the designated method in code.

For the threshold schemes (IDA and secret sharing) $k = 2$ and $m = 1$. But, to a fair comparison, for the information replication method we assume $n = 2$ and $k = 1$. The storage efficiency corresponds to the percentage of usable space, tolerating only one CSP failure.

	Storage efficiency	Computational complexity	Implementation complexity	m failures tolerance	Minimum n clouds
Secret Sharing	33%	$(n \log^2 n)$ [28]	high	$m = n - k$	3
IDA	66%	$O(n \log n)$ [26]	high	$m = n - k$	2
Secret Sharing+IDA	66% [1]	$(n \log^2 n)$ [28]	very high	$m = n - k$	3
Replication	50%	N/A	very low	$m = n - 1$	2
Simple Parity	66%	$O(n)$	low	$m = 1$	3

Table 2.1: A comparison between the solutions presented in this section. All the methods apart from Secret Sharing are combined with a symmetric cipher. Schemes that use secret sharing must respect $n = 2k - 1$

Analyzing Table 2.1 we can conclude that there are only three methods(IDA, Secret sharing plus IDA and Simple Parity) that we can use to met our objectives of low storage overhead and decent performance discussed in Section 1.2. Being the easiest one to implement the simple parity method.

The solutions that use secret sharing do not require the storage of a cryptography key locally in the device. All the others have that requirement as they are combined with a symmetric cipher that uses a key stored in the application configuration file.

2.5 Storage Integrity

A storage system that provides availability and confidentiality can not be reliable if it has no integrity mechanisms. If we can not verify that the data we are accessing is the same we had stored before, we can have a serious security problem. Imagine that some attacker manages to inject some malicious code into a stored file, if we do not have a mechanism to detect that the data has been altered, in the best case the file will only be corrupted, in the worst case we will infect the user device with the injected malware.

As it is impossible to prevent all causes of integrity violation, we aim to have a data storage

[1] A little bit less than 66% because of the additional information of the encoded symmetric key 2.4.2

system that can detect and tolerate them.

In cryptography context, a Message Authentication Code (MAC) is a piece of information used to authenticate a message, and sometimes it is incorrectly called a cryptographic hash function since a cryptographic hash function is only one of the possible ways to generate a MAC. They accept as input a secret key and an arbitrary-length message, and gives as output a MAC value that protects both the message data integrity as well as its authenticity, by allowing someone that also has the secret key to detect any changes to the message content [30].

As said before one way to generate a MAC is by using cryptographic hash functions. Hash functions are sometimes called compressing functions, that is because they generate from an arbitrary size data a fixed sized data (resuming the original data). They accept as input an arbitrary-length message and return a value called hash value, hash code, digest, or simply hash.

A hash function must have three essential characteristics:

Pre-image resistance Given a hash value h, it should be computationally hard to find any message m such that $h = hfunc(k, m)$, where k is the hash key;

Second pre-image resistance Given a message m_1, it should be computational hard to find a different message m_2 such that $hfunc(k, m_1) = hfunc(k, m_2)$, where k is the hash key;

Collision resistance Given two messages m_1 and m_2, it should be computational hard to find a hash such that $hfunc(k, m_1) = hfunc(k, m_2)$, where k is the hash key.

A hash function that respects those rules can be used for integrity checks because they represent a unique resume of the original data using that function. We can calculate the hash value of a message, send it along the message, and the receiver then using the same hash function generates the hash value of received content and if it matches the message has not been modified.

Most CSP provide hash values of the stored data in metadata fields. However, this is not enough to enforce the integrity of our system. An attacker can modify the file, generate a new hash value and change the meta-data. That leads us to the initially exposed MAC, which requires a secret key along with the data as input. In that way, the attacker would need the secret key to generate a hash with the same value.

2.6 Related Work

Our solution is unique for the Android platform, some applications combine multiple CSP accounts into one, basically merging the available space into one file system but those do not add any security features or redundancy features. There are some that allow users to protect data confidentiality by encrypting the files before storing them remotely, but again they do not provide availability mechanisms. Also, our system allows users to mount the FS in a local folder, integrating it into the system.

Bellow we describe some solutions that somehow are related to our solution:

TOPDOX Ficheiros & Cloud Docs [31] TOPDOX is an Android application that helps its users to manage documents stored in the cloud. They have a feature that allows users to merge several CSP into one FS, but the similarities with our solution end there, they do not provide availability, integrity or confidentiality mechanisms.

EasySSHFS [32] EasySSHFS is a solution that, like ours, allows users to mount a remote FS into a local storage folder, but it is not a cloud storage solution and does not add any security or redundancy mechanisms. The Android application allows mounting a remote FS using SFTP that most SSH servers support and enable this SFTP access by default.

SafeCloud Photos [33] Safe cloud is a solution very similar to ours, but it only works for photos. It is an android application that allows the users to configure public CSP accounts and then the photos are split and stored across them using cryptographic techniques to ensure that if an account is compromised, there is no leaked information about the stored photos. This solution is focused on privacy, and does not provide any redundancy mechanisms, that is another thing that differentiates this solution. Our solution can be considered an extension of this solution, allowing any file extension and adding availability and integrity mechanisms.

Although there is no implementation of a cloud file system with improved confidentiality, integrity, and availability for Android devices, there are some implementations for the Linux OS that meet those requirements:

DepSky [34] DepSky is a system that improves confidentiality, integrity, and availability of

data stored in the cloud, it uses encryption, encoding, and replication of data over several commercial clouds.

The goal is similar to ours, but they use different encoding and cryptographic techniques, they operate at a lower level and we at the file level, their system behaves like a virtual disk where we can store blocks. They also use commercial clouds instead of free ones and aim to store critical data such as medical records and financial summaries.

We can see the architecture of DepSky in figure 2.14, where the clouds are accessed using their standard API and DepSsky algorithms are implemented as a software library in the clients.

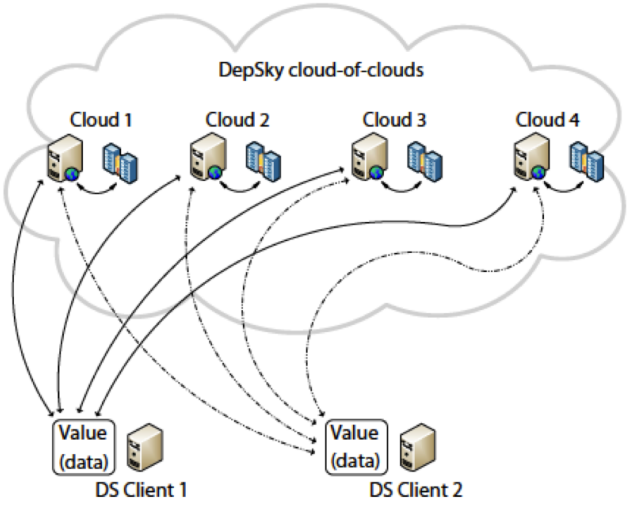


Figure 2.14: DepSky architecture with 4 clouds and 2 clients [34].

They have two protocols, the DepSky-A to improve availability, that replicates all the data in clear text in each cloud. And DepSky-CA to improve availability and confidentiality, that uses secret sharing and erasure code techniques to reproduce the data in a cloud-of-clouds.

In DepSky-CA with four clouds, first is generated an encryption key, and after that the original data block is encrypted. Then the encoded data block is erasure coded and is computed key shares of the encryption key. In this case, we get four erasure coded blocks and four key shares because we use four clouds. Lastly, is stored in each cloud a different coded block together with a different key share, this is one of the discussed techniques in section 2.4.

C2FS [35] C2FS is a system that uses DepSky as data storage service, but implements a POSIX

compatible file system. It improves the data and metadata availability, integrity and confidentiality. The architecture of the system is the same as DepSky, and it stores the data in a cloud of clouds. C2FS offers a distributed directory service using a coordination service, named DepSpace, to store the system meta-data.

The major advantages over DepSky are the ease of use (high level of abstraction, it behaves like a regular local folder) and distributed directory services that ensure the meta-data availability and confidentiality.

Chapter 3

System Architecture

In this chapter, we discuss the general architecture components from a high-level perspective.

Our system is meant to run in Android smartphones and allows the users to have access to files stored in the cloud through a local folder. When the user saves a file in that folder, the file is not stored locally in the device but sent to the cloud, and when the user requests a file, it is downloaded from the cloud and presented to the user. However, instead of just storing the files in a CSP, we perform some modifications in order to keep them safer while in the remote space. And for safer we mean that even if one provider is hostile, goes out of business or an attacker manages to access or corrupt our files, there is no loss of data or confidentiality.

In figure 3.1 we can see the system architecture and how its components interact. We have a FS implementation that intercepts the Android FS calls and replaces them, and some modules that help us to achieve file availability, confidentiality, and integrity. The Integrity module is used to verify that an unauthorized entity has not modified a file that we previously stored. The Redundancy module is responsible for giving the FS the ability to resist to CSPs downtime, by adding redundant information to the files and storing them split across various CSPs. The Confidentiality module is used to encrypt the files, so they remain confidential even if they are leaked. There are also two other modules, the Cloud Abstraction module, that is responsible for making the communication between our system and the remote space (CSPs), and the Cache module that is used to improve the system performance maintaining local copies of some files.

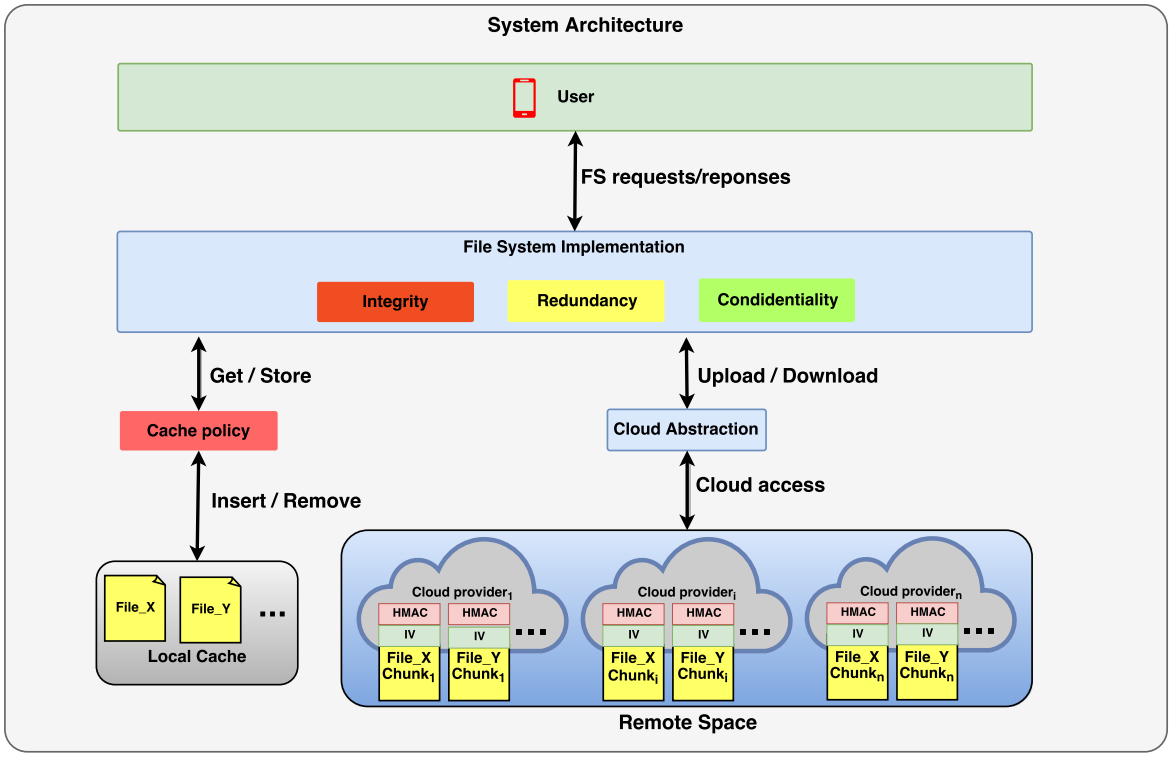


Figure 3.1: Presented solution System Architecture, with a configuration of three CSPs

3.1 Cloud Abstraction Module

This module is necessary because each CSP has its own API to manage the remote space, so we define an intermediate API that allows us to abstract from the individual CSP APIs and performs actions uniformly across the different CSPs.

We have a generic interface with the necessary functions (store a file, get a file, list the files in a folder, among others) to interact with a CSP and independent implementations to interact with different CSPs. Then when calling a function, it automatically calls the correct implementations depending on the CSP we are interacting with.

3.2 Availability Module

In order to improve the availability of the stored files, we take advantage of redundancy techniques. This module provides a series of functions that allow us to add redundant information to the system and recover lost data.

As every redundancy technique implies a performance overhead and a higher usage of storage space we need to choose a method that can balance those two aspects. Our system uses coding techniques because as seen in detail in section 2.2 the alternative (information replication) would have a much higher storage space overhead. Among the coding techniques, our system uses simple parity, and that is because it is easy to implement, has a low impact on the system performance as it is based on simple operations. The downside of this technique is that it only provides fault tolerance of one CSP at a time, but as the probability of more than one become inaccessible (or the files become corrupted) at the same time is low (0.00001% as seen before in Section 2.2), it is a good choice.

In order to add redundant information to the system, before storing the files in the remote space, we split them in $N-1$ chunks, where N is the number of CSPs in use, and then generate an extra chunk that stores the redundant data (calculating the bit by bit parity between the other chunks). As a result, we have N chunks, one for each CSP as we can see in figure 3.2.

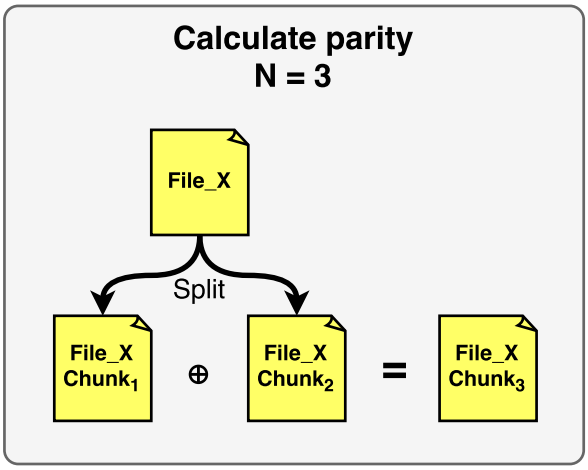


Figure 3.2: Calculating redundant information of a file using simple parity, with a configuration of three CSPs

When all CSPs are accessible we can recover the file by getting the first $N-1$ chunks, but to recover a lost file chunk (due the CSP being inaccessible or the data being corrupted) we need to get the $chunk_N$ (the one with the redundant data) and recover the missing chunk by calculating the bit by bit parity between the available chunks.

Summing up, we use this module to calculate the parity between N input chunks, generating a new chunk of the same size. This can be used to generate a redundant chunk or to recover a missing one.

3.3 Confidentiality Module

Files remotely stored in CSPs are protected by a security mechanism (Authentication by Username and Password), and some CSPs claim that the files are encrypted, but we do not know who has access to the cryptographic keys, so we assume the worst case scenario where the CSPs can be malicious. Also, the authentication mechanism is not enough to ensure the confidentiality of the stored files, and there are many examples where this mechanism was violated, as we presented before in Chapter 1. Some Redundancy techniques also provide a low level of confidentiality, as shown in Section 2.2, but we want a robust system able to maintain the confidentiality of all the stored content even if the entire content of a single cloud is leaked.

As we can not trust in the CSPs to keep our files confidential, we use cryptographic techniques to achieve that.

We opted for symmetric cryptography that requires only one key. That key must be known by the sender and the receiver, as the encryption and decryption process relies on that key, if the key is revealed, the data can be revealed too. Symmetric Cryptography is much faster and energy efficient when compared to Public Key Cryptography. This subject is discussed in more detail in Section 2.2.

Within symmetric cryptography there are some alternatives that can be used [36]. We opted for Advanced Encryption Standard (AES) cipher because it is widely accepted is the most utilized in the industry but most important because it is considered secure. Also some smartphones chipsets support AES hardware acceleration. We strongly recommend the use of that cipher, but there are no impediments to the utilization of another symmetric cipher, as long it is considered secure and resistant to cryptanalysis.

The AES cipher has many operation modes, but we recommend Cipher Block Chaining (CBC) mode where we need a key and an Initialization Vector (IV). With CBC the previous ciphertext block (or IV) is effectively random (and independent of the plaintext block), being the block cipher an effectively random string.

AES comes with three standard key sizes (128, 192 and 256 bits). The 256-bit version is a bit slower than the 128-bit version (by about 40%) but as NIST recommends the 256-bit version over 128-bit version we follow that recommendation. The key is the same for all files and is generated

from a user-provided password in the initial configuration of the system.

The IV is used to add randomness at the start of the encryption process. When using CBC mode (where one block incorporates the prior block), we need a value for the first block, which is where the IV comes in.

If we had no IV (or use the same) using CBC with just a key, two files that begin with identical text would produce identical first blocks. If the input files changed midway through, then the two encrypted files would start to look different beginning at that point and through to the end of the encrypted file. If someone noticed the similarity at the beginning, could use that information to determinate the key or the beginning of some files throughout cryptanalysis. Because of that, we cannot use the same IV for different files, and this is very important to keep the process secure.

Confidentiality is obtained by encrypting the files before storing them in the CSPs. The key used for encryption is the same for all files and is generated from a user-provided password, the IV is randomly generated for each file, and changed every time it is updated. The IV is then stored in the header of the files as it is needed for decryption.

3.4 Integrity Module

For achieving data integrity, we use a MAC that is a piece of information used to authenticate a message. A MAC accept as input a secret key and an arbitrary-length message, and gives as output a MAC value that protects both the message data integrity as well as its authenticity, by allowing someone that also has the secret key to detect any changes to the message content [30].

In Figure 3.3 we can see how this method can be used in our system to protect the file integrity, giving us the ability to detect violations. The process is simple: we just calculate an HMAC value for each chunk of the file (one for each CSP) and store it in the file header before sending it to the CSPs. After retrieving a file we calculate the HMAC value again and check if it matches the one in the header. This operations have an delay associated that is the time to do the calculations and comparisons, but the recent hash functions have a great performance even when dealing with great amounts of data.

The generation of HMAC values can be made using different hash functions. We recommend

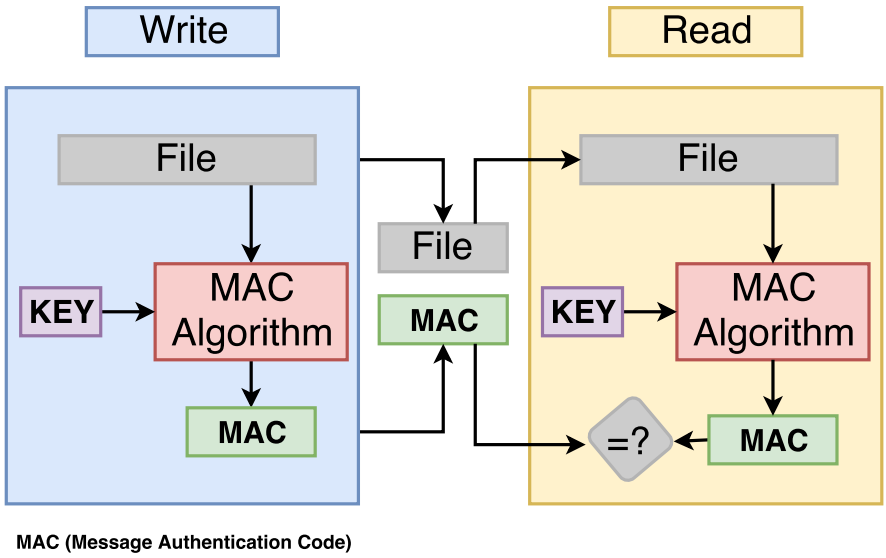


Figure 3.3: HMAC calculation at the moment of dispatch and verification at the moment of recipt

the usage of Secure Hash Algorithm (SHA) 2 because it is widely used and it is defined as a standard by NIST which means that the algorithm validity has been thoroughly tested. There are many attack attempts against the algorithm but none completely compromises its security [37]. NIST already defined a new standard to replace SHA 2 version but for now there are no available implementations and SHA 2 is still considered secure to use.

The key used in the HMAC value generation is the same for all files and is generated from a user-provided password in the initial configuration of the system.

3.5 Caching Module

We use a caching system to keep some files in the local storage because our files are stored remotely across several CSPs, and we have a delay to download them. This module is not mandatory, but it can improve a lot the performance of the system.

Usually to implement a cache mechanism we have access to both client and server. We can maintain a cache in the server, the client, or both, but as our system uses services that are SaaS, we can not run any code on the server. So we are restricted to client caching, and the caching structure is limited to file level (we could have block level or file level), as is the data unit provided for the services we use.

There two main things that we must be aware when developing a cache mechanism, the cache replacement policy, that defines how the cached content is replaced when the cache is full, and the writing policy that defines when the cached content is written to the backing storage (the clouds in our system).

Examples of writing policies:

Write-back When a file is modified it is written only to the cache. Then it is only written to the backing storage (the clouds) when a cached file is about to be modified or replaced. We can define other policies to trigger the write to backing storage;

Write-through The modifications to files are done synchronously to local storage (cache) and the clouds;

Write-back with on close policy When a file is modified, instead of sending the modifications synchronously, we just store it in the local cache and write it to the clouds only when the file is closed.

The one that is most adequate for our system is Write-back with on close policy because it reduces the number of writes. This has a huge performance impact, since in this case writes involve network operations.

One important thing about our cache system is that the files are stored in clear text, they are not encrypted in the cache in order to speed up the retrieval of cached files. After getting a file from the clouds, we decrypt it first and only afterwards store it in the cache.

When it comes to cache replacement policies, we use a Least Recent Used (LRU) cache policy, which, as the name implies, consists of removing the least recently used items first. This cache policy is known for working well in workstations file systems [38]. The reason for its success is that it assumes that blocks which have been used in the recent past will likely be referenced again in the near future, a property known as temporal locality. There are other options that could be used but as LRU is easy to implement and is known to work well in most cases we use it in our system.

When using a caching system, we must be aware of possible inconsistent states of the file system, and this can be a huge problem if we consider a scenario where we have multiple parties accessing

and modifying the file system contents. Although our system is designed to operate in a scenario where only one device changes and accesses the data, there are still some possible inconsistent states that we need to take into account:

- Write failure (to the clouds)
- Critical application failure

To reduce the probability of an inconsistent state we can use a simple version manager when updating a file in the file system, only deleting the old version stored in the clouds when we get a confirmation of all chunks submissions to the clouds.

3.6 General Overview

This section shows how all the modules collaborate to achieve the proposed objectives making the remote file storage more secure.

In Figure 3.4, we present the contribution of each module in the process of storing a new file in our FS. The user pastes a new file in our FS folder. Then, we create a file in the local cache and perform the writes locally, then when a flush is called, we tell our Cache module to store the file (removing other files if necessary), and call the Confidentiality module to encrypt the file using a key generated using a previously provided user password and a randomly generated IV. The next step is to split the file in N-1 chunks (eg. N=3) and calculate an extra chunk by invoking the Redundancy module. Then for each chunk, we use the Integrity module to calculate a HMAC, and store in the header together with the previously generated IV, finally we call Cloud Abstraction Module to store the files across the N CSPs.

When accessing a file, first we check if we have in the cache. If not, we proceed to download N-1 chunks and check their integrity, by comparing the HMAC value of the download chunks with the HMAC stored in the header. Then we join the chunks (removing the headers first), obtaining, as a result, the encrypted file, now we just use the IV previously stored in the header and the key to decrypt the file. Finally, we store the file in the local cache, so the subsequent accesses to the same file do not require networking operations. This process is illustrated in Figure 3.5, and represents the best case scenario where the file integrity was not violated, and all CSPs

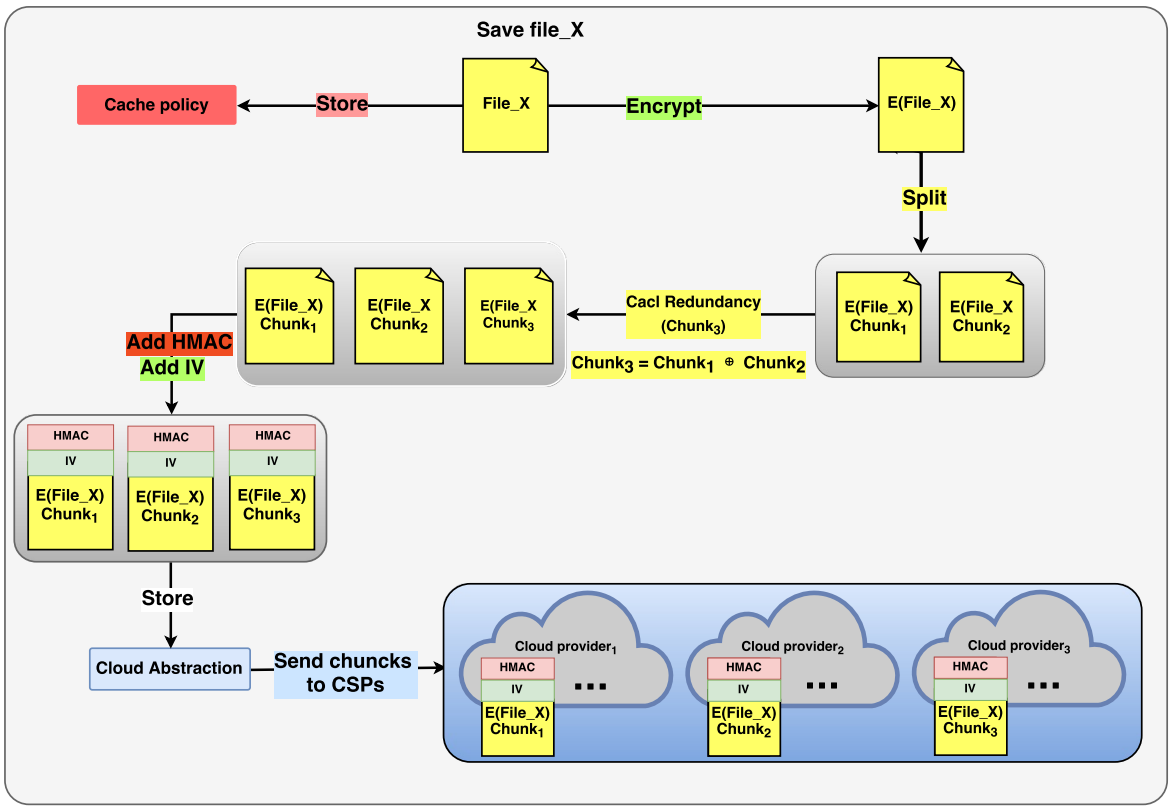


Figure 3.4: Storing a file, with a configuration of N=3 clouds, where the redundant chunk is stored in cloud number 3

are available. To best understand what happens in a situation where a failure occurs look into Figure 3.6, that represents the FS work-flow when one of first N-1 CSPs are unavailable.

If after getting a chunk from a CSP we detect that its integrity has been violated we just download a chunk from other CSP, and proceed as if the CSP was off-line, as we can see in Figure 3.6. Afterwards we delete the invalid chunk, calculate it again using the other chunks and send it to the corresponding CSP.

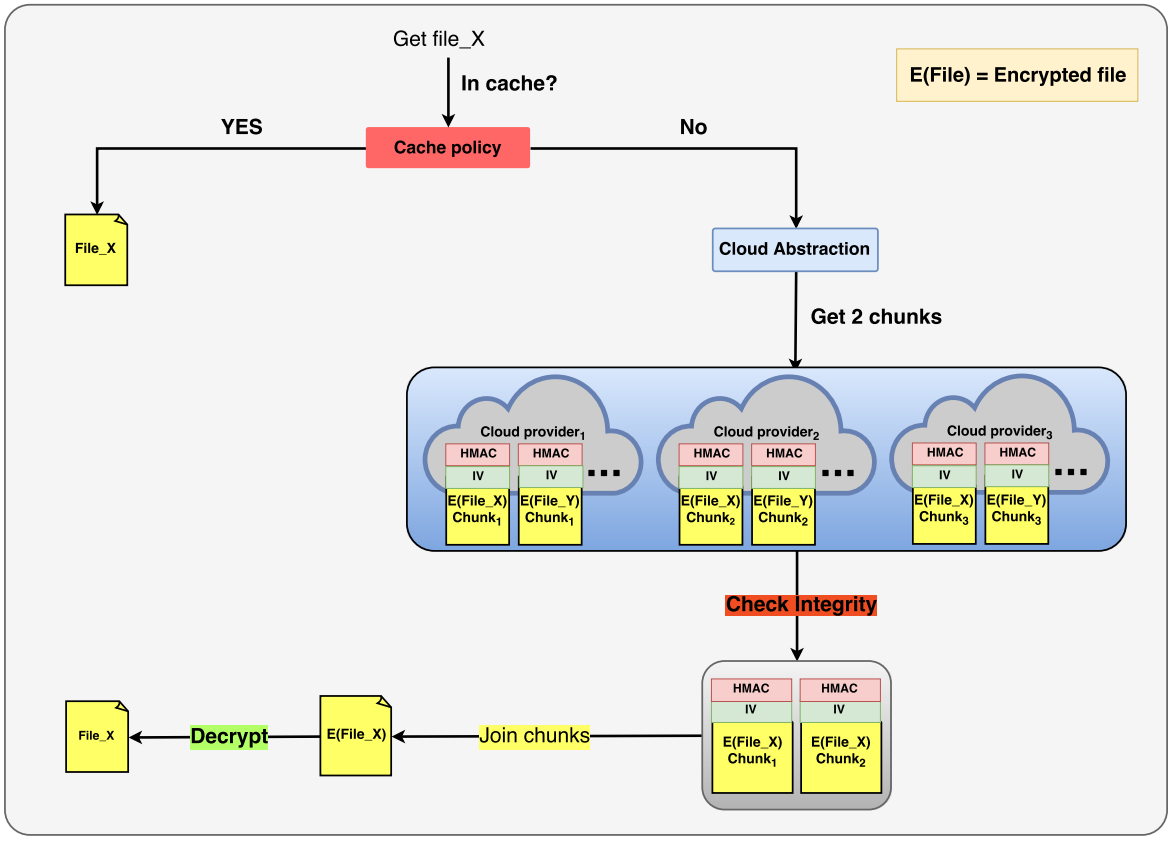


Figure 3.5: Getting a file, with a configuration of N=3 clouds, where the redundant chunk is stored in cloud number 3. All clouds available.

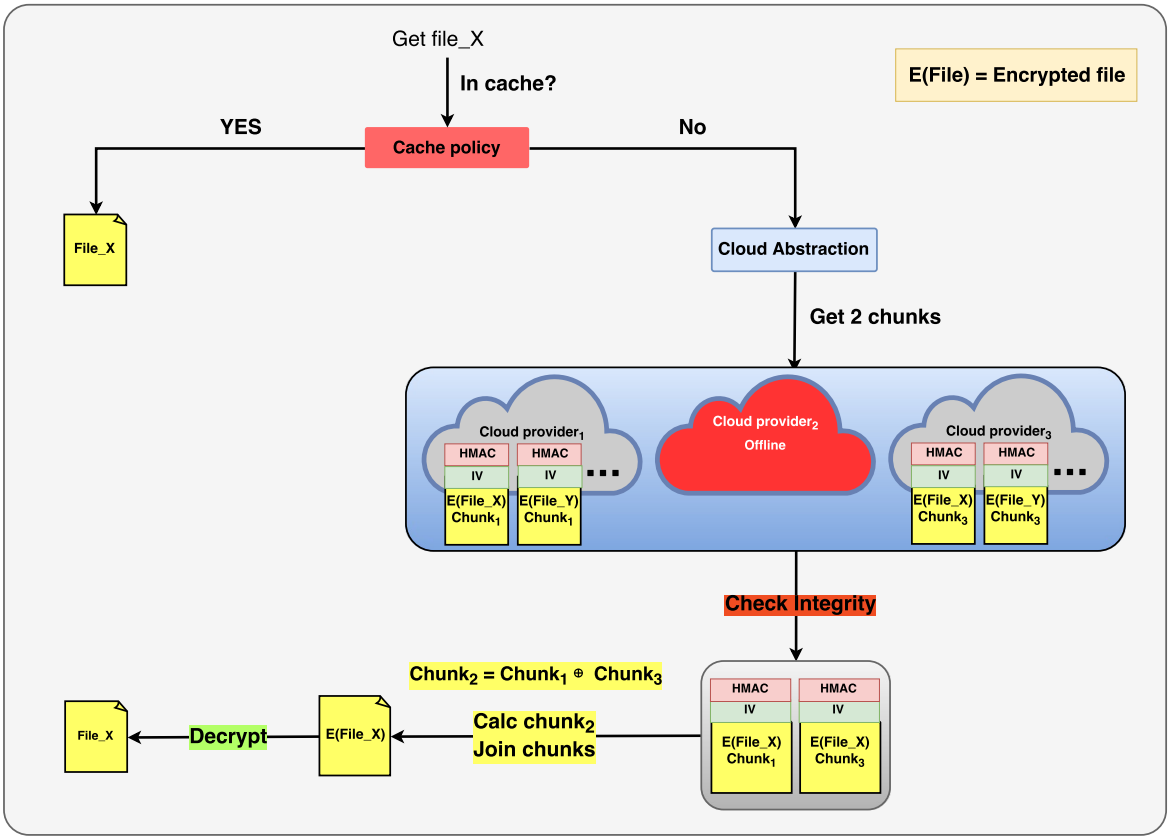


Figure 3.6: Getting a file, with a configuration of N=3 clouds, where the redundant chunk is stored in cloud number 3, cloud 2 offline.

Chapter 4

System Implementation

In this chapter we describe how the system implementation is structured, detailing how each part of the system is built. The base of all the system is the Linux kernel module File System In User Space (FUSE) that is present in the Android OS and provides an interface for userspace programs to export a FS to the Linux kernel. Besides the FUSE implementation, this work consists in a development of an Android application that allows the user to manage the settings of the system.

4.1 Implementation Structure

There are two distinct components in our system, the FS implementation and the Android application. The FS implementation is the main component of this work as is there that we implement all the security features. The Android application is used to generate the settings to be imported to the FS implementation and to manage it, allowing the user to mount and unmount the FS.

To do the FS implementation, we use the library libfuse that provides a reference implementation for communicating with the FUSE kernel module. The result is a standalone application that links with libfuse that provides functions to mount the file system, unmount it, read requests from the kernel, and send responses back. This means that with libfuse the kernel redirects the requests to our application allowing us to implement the integrity, redundancy and confidentiality methods previously discussed in Chapters 2 and 3.

The Android application is necessary to simplify the usage of the system. Without it the system can run, but we need to use a command line to execute it, and manually generate the access tokens to the CSPs. It provides a clean interface that simplifies those steps.

4.1.1 Limitations

The main limitation of the implementation is that it only works on Android devices with root access [1]. We need root access to use FUSE that is an initial requirement of the proposed work. In the regular Linux kernel we have access to fuse from the applications, but in Android, as the kernel has some modifications this access is revoked, so in order to have access to it we had two options: build a custom ROM (custom version of the Android OS) with modified Security-Enhanced Linux (SELinux) permissions or request root access. We have opted for the second option because build a custom ROM is too much time consuming and a bigger requirement than just asking for root access.

We also needed to design the implementation having in mind that it needs to be done using the programming language C that is the only viable option to implement fuse in Android. Due to the low-level implementation of the system, each module was validated using unit tests and a memory leak check tool (valgrind) in order to prevent implementation errors.

4.2 FUSE

Normally in order to create a custom FS, we would need to write a kernel module as it manages the file manipulation requests, but that would be hard to achieve and would require the development of a custom ROM. In order to overcome the complexity of that, we have FUSE that allows the development outside of the kernel level (at the user level).

The FUSE project has two components: the fuse kernel module and the libfuse userspace library that provides the reference implementation for communicating with the FUSE kernel module. As we do not actually work with the kernel module, we will not go into detail about how it works. For more information about it consult the documentation [39].

[1] Android uses the Linux kernel, rooting an Android device gives the user similar access to administrative (superuser) permissions just as on Linux.

Figure 4.1 shows an example where the file system is mounted at $/tmp/fuse$ and the user program is $./hello$. When a request is made at the mount point ($ls - l/tmp/fuse$) the request is passed to the Virtual File System, then passed to the FUSE kernel module and finally to libfuse that calls an appropriate function (in this case $hello\_readdir()$, called every time that the OS wants to read the contents of a dir), then a response is generated and goes throughout the same path back.

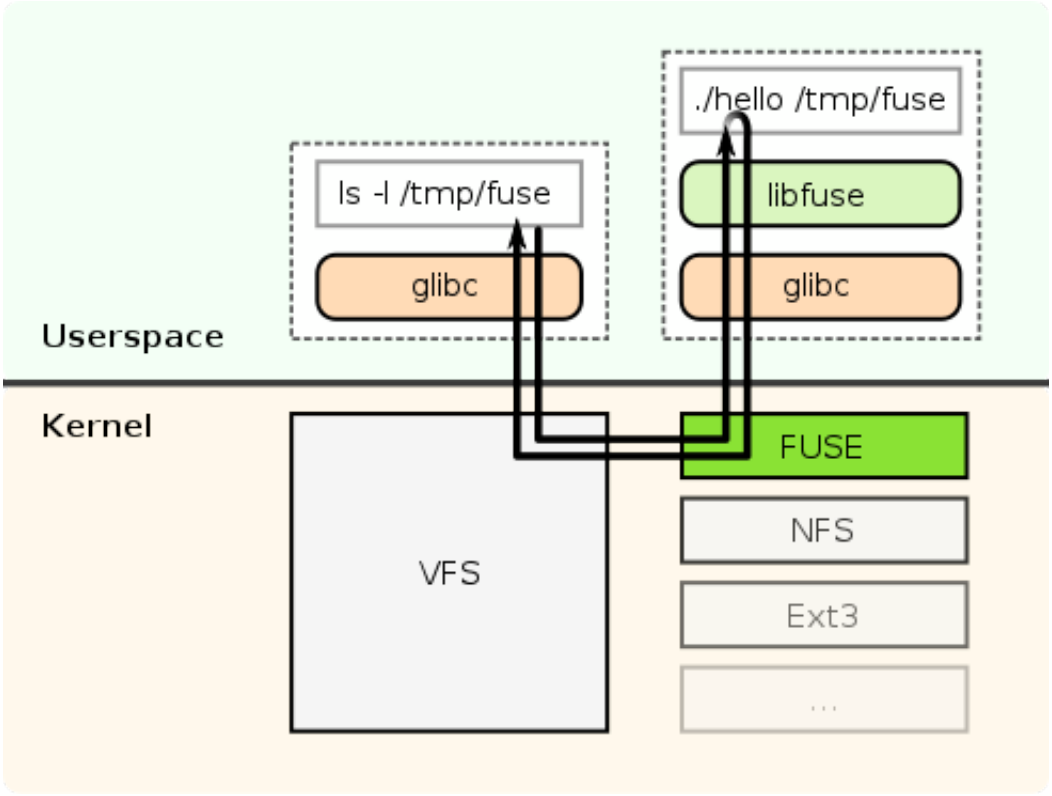


Figure 4.1: A diagram that shows how FUSE works, the arrows show the OS components that the requests and responses must go troughouth [40].

In the libfuse component when the user mode program (our implementation) calls $fusermount()$ it parses the arguments passed to the program, then calls $fuse\_mount()$ that creates a UNIX domain socket pair, then forks and execs $fusermount()$ passing it one end of the socket in the $FUSE\_COMMFD\_ENV$ environment variable. $fusermount()$ loads the fuse module then open $/dev/fuse$ and send the file handle over a UNIX domain socket back to $fuse\_mount()$. $fuse\_mount()$ returns the filehandle for $/dev/fuse$ to $fusermount()$. The function $fusermount()$ calls $fuse\_new()$ which allocates a struct $fusedatastructure$ that stores and maintains a cached image of the FS data. Lastly, $fusermount()$ calls either $fuse\_loop()$

or $fuse\_loop\_mt()$ which both start to read the FS system calls from $/dev/fuse$, calling the usermode functions that we have implemented. The results of those calls are then written back to the $/dev/fuse$ file where they can be forwarded back to the system calls.

In order to have a functional FS, we do not need to implement all FUSE functions. Due to the lack of time we decided to implement only the essential ones, which we can see in listing 4.1. All functions return an integer 0 or a positive number for success, or a negative value selected from $errno.h$ if there is an error.

```
static struct fuse_operations mcfs_oper = {
        .getattr    = mcfs_getattr,
        .readdir    = mcfs_readdir,
        .unlink     = mcfs_unlink,
        .open        = mcfs_open,
        .read        = mcfs_read,
        .flush      = mcfs_flush,
        .write = mcfs_write,
        .create = mcfs_create,
        .release = mcfs_release
};
```

Listing 4.1: fuse_operations struct containing only the functions that we have implemented

The functions implementation should have a prefix related to the FS name. For example, in an SSH FS we should use $ssh\_getattr$, ssh_read and so on. In our case, we use the prefix mcfs (Multi-Cloud File System).

Most of the FUSE functions have two ways to identify the file being operated. The first, which is always available, is the "path" argument, which is the full pathname (relative to the FS root) of the file in question. The second is a "file handle" in the $fuse\_file\_info$ structure. The file handle is stored in the fh element of that structure. If we want to use it, we need to set it in the $open$, $create$, and $opendir$ functions so other functions can then use it.

4.2.1 Main function

This is the function used to initialize the file system by calling the $fuse\_main()$ function and passing it the implemented fuse operations, but we can use it to generate a shared "object" ($fuse\_file\_info$ structure) to be accessed in any operation that we use to store information about cache and other useful things. First, we need to parse the program parameters in our $main()$ function. Four parameters are required: the mount point path, the cache size, the password for confidentiality and the password for integrity.

After parsing the arguments we use them to fill a struct called fs_state that as we can see in Listing 4.2 has the following parameters:

$FILE * logfile$ A File pointer to a logfile where we print errors and warnings;

$char * rootdir$ A string containing the parent directory of the mount point;

$unsignedchar * conf\_key$ The cryptographic key used in to encrypt the files;

$unsignedchar * int\_key$ The cryptographic key used in to generate HMACs;

$structcache\_info * cache$ A struct that maintains the cache state;

$structhashmap * journal$ A struct that maintains the journal state;

$pthread\_mutex\_tmetadata\_cache\_mutex$ pthreads mutex used to protect the cache struct from concurrent modifications ;

$pthread\_mutex\_tjournal\_mute$ pthreads mutex used to protect the journal struct from concurrent modifications;

$pthread\_cond\_tjournal\_condition$ pthreads condition used to signal and wakeup the worker thread.

This struct is important as it represents the FS state. We can access at any time in the FS operations by calling $fuse\_get\_context() -> private\_data.$

Besides the program arguments described we also need a config file that must be in the same directory as the program with the name $config.json.$ This config file is generated by our Android

application as described in section 4.11. This file has the access tokens that allow the system to communicate with the CSPs.

```
struct fs_state {
    FILE *logfile; //File pointer to error log file
    char *rootdir; //mount point parent dir
    PNODE provider_list; //A list of providers, each PNODE has information
        about a provider and a pointer to the next PNODE
    unsigned char* conf_key; //confidentiality key
    unsigned char* int_key; //integrity key
    struct cache_info* cache; //a struct that maintains the cache information
    struct hashmap* journal; //a struct that maintains the journal
        information
    pthread_mutex_t metadata_cache_mutex; //mutex to lock the access to the
        cache hashmap
    pthread_mutex_t journal_mutex;          //mutex to lock the access to
        the journal hashmap
    pthread_cond_t journal_condition;      //condition to wakeup the journal
        thread
};
```

Listing 4.2: fs_state struct implementation

After having our fs_state struct filled with the necessary information we call $fuse\_main$, passing that structure so we can access it inside the FS operations. We also pass the structure $mcfs\_oper$ to $fuse\_main$. The $mcfs\_oper$ structure contains the functions of our FS as shown before in listing 4.1, and the detailed implementation of those functions is described in section 4.9.

4.3 Availability module

In this module, we implement the necessary functions to add redundant information to our FS. The objective of having redundant information in the system is to be able to recover from an eventual loss of access to a CSP. So we need to be able to add redundant information to the system and be able to recover files from the redundant information.

As discussed before in chapter 3 in order to generate redundancy we use parity checks, we split

the files into n (being n the number of CSPs - 1) chunks and then generate an extra redundant chunk. And to recover a file we get $n - 1$ chunks plus the redundant one and calculate the missing chunk. For that, we have created three functions: one that splits a file into n chunks, one that joins n chunks of a file, and one that calculates from n chunks an extra one calculating the bit by bit parity between them.

The function used to calculate the parity between n file chunks is used both to generate the redundant chunk or to recover a missing chunk as the logic is the same.

4.4 Confidentiality module

This module has the necessary functions to encrypt and decrypt the files stored in our FS with the objective of keeping the confidentiality of the files even when a CSP is hostile or an attacker manages to get access to the entire content of a CSP account.

As discussed in detail in Section 3.3 we use the algorithm AES-256 in the encryption/decryption process. As implementing the whole algorithm is too much time expensive and often leads to programming errors we use the OpenSSL EVP library that provides a high-level interface to cryptographic functions.

We created a function to encrypt or decrypt a file depending on the value of a parameter $should\_encrypt$. It requires a key and a IV to be used in the process. The file to encrypt/decrypt is passed as an argument, and the produced output is stored in a path also passed as an argument.

The cryptographic key used in decryption and encryption is defined in the initial configuration and derived from a password. We defined a function that receives an alphanumeric password and generates a cryptographic key with a size of 256 bits (as we are using AES-256) using OpenSSL EVP module functions with that objective.

The IV value is critical because if we repeat it or make its generation predictable, an attacker could do cryptanalysis and possibly reveal part of the original file or the cryptographic key. In order to avoid that, we generate a random IV for each remote write, not only for each file. When we update a file in the remote space we generate a new IV. For that, we created a function that generates and returns a fixed size random IV.

Apart from the above-described functions, there are additional ones. One is used to initialize the crypto library and the other to do the cleanup by freeing the used resources.

4.5 Integrity module

This module is used to give the system the ability to detect integrity violation of the stored files for that we use HMACs generated with the SHA algorithm as described in Section 3.4. We had two options on this case:

File level Integrity Generate a HMAC for the files before the redundancy steps (split and generate a redundant chunk), store it in the head of all chunks and then store them remotely. When accessing a file, we would check if all chunks have the same HMAC and if the HMAC of the reconstructed file (redundancy module again) matches the file. If any of the checks fail then we can assume that the integrity of the file has been violated.

Chunk level Integrity Instead of just checking the integrity of the whole file we can check the integrity of each chunk. For that we can generate a HMAC for each chunk instead of only one HMAC for the original file. Then when accessing a file we can detect the integrity violation of separated chunks and get the redundant chunk if necessary (if the integrity of a single chunk is violated) to reconstruct the original file.

We have chosen to implement chunk level integrity, as it has the benefit of maintaining the access to a file when a CSP account is compromised, and its stored files lose the integrity. For that we have implemented two functions: one that generates a HMAC for a given file using a given key, and one that calculates the HMAC of a given chunk and compares it with a given HMAC that is used to verify the integrity of the stored files.

The key used in the HMAC generation process is generated using a user-provided password in the initial configuration and is derived using the same function used in the confidentiality module to generate the cryptographic key as they have the same size (for confidentiality we are using AES-256, for integrity we are using SHA-256).

4.6 Cloud Abstraction module

To interact with the CSP accounts which are our backing storage, we must create an abstraction module because every CSP has its proprietary API and we want a CSP independent implementation. For that, we defined some generic functions that make calls to different CSP APIs based on the CSP we are accessing in the moment of the call.

In this work we only implemented the access to the DropBox API. This is enough to demonstrate the validity of our solution if we use different accounts on that CSP.

We have defined the following functions:

$auth\_init$ Used to initialize the CSP client, performing the authentication process using the initially provided authentication tokens;

$read\_dir$ Used to read the contents of a remote directory, returns an array with the metadata of the files in a folder passed as argument;

$get\_metadata$ Used to get the metadata of a specific file passed as argument;

$download\_file$ Used to download a file passed as argument;

$upload\_file$ Used to upload a file passed as argument.

4.7 Cache module

The cache module is used in our system to improve the overall performance of the read and write operations, as these operations are time-consuming due the network accesses needed to store and get the files from the cloud. We have implemented a LRU cache substitution policy that prioritizes the files utilized in the recent past, and discards the least recently used ones when freeing space is necessary.

We have defined a struct ($cache\_info$, shown in Listing 4.3) to represent the cache. It maintains in memory all the necessary information about the cache, including the cache directory where we store the files, and the used space. In this struct we store a hashmap containing $metadata\_entry$ structs 4.3 where using the path of the files or folders in the FS as a key.

When we list the contents of a folder (when we access it), we add a new $metadata\_entry$ to the $metadata\_cache$ hashmap with the variable $isDir$ set to true and with the variable $contents$ (an hashmap) filled with the files inside that folder. We maintain this in the cache in order to improve the performance of the folder access. This way we do not need to request the the folder contents from CSPs every time we access the folder. We also add the metadata of the files in the folder to the cache adding a $metadata\_entry$ to $metadata\_cache$. Note that when we do this, we do not have the file in cache, we are only caching the metadata. Later when we store a file in the local cache we add the path in the local storage to the $cache\_path$ variable of correspondent $metadata\_entry$.

The metadata of a folder is invalidated every time a file of that folder is remotely updated and when we create or delete a file, and the metadata of a file is invalidated every time it is remotely updated or deleted.

```
struct cache_info{
    long cache_size; //total size of the cache
    long used_space; //used cache space
    char* cache_dir; //caching directory
    struct hashmap* metadata_cache; //cached files
};

struct metadata_entry {
    double st_size;       //file size
    time_t atime;         //access time
    time_t mtime;         //modify time
    bool isDir;           //true for directories, false for file
    bool local_cache;     //true if available in local cache
    bool remote_updated; //true if updated in the clouds
    char* cache_path;     //path for the cached file (if available)
    struct hashmap* contents; //the contents of the folder(only for folders)
};
```

Listing 4.3: Cache representation structs function

In order to make the cache mechanism work, we have defined the following functions:

get_cached_attr Used to get the cached metadata of a file;

add_cache_attr Used to cache the metadata of a file;

$add\_folder\_contents$ Used to cache the contents of a folder;

$remove\_folder\_contents$ Used to remove the cached contents of a folder;

add_file_cache Used to add a file to the cache, here we also remove any necessary files using LRU replacement policy;

$remove_file_cache$ Used to remove a file from cache, deleting it from the local storage.

The LRU cache replacement policy is implemented in the function add_file_cache, when adding a file to cache we remove a file if necessary according to the policy (described in detail in Section 3.5). In order to determine which files need to be removed we use the $atime$ variable that represents the instant of the last access to the file.

4.8 Journaling module

In order to better recover from an eventual inconsistent state of the FS, we started the development of a journaling module, to maintain a journal with pending operations (not yet done in the remote storage). With this module we can also allow the users to operate the FS without Internet access, enabling them to change files that are in the local cache, adding them to the journal and only updating the data in the CSPs when connectivity is restored.

This was done by instead of doing the remote operations right away, add them to an hashmap, defined in the fs_state struct (see Listing 4.2), and creating a worker thread that waits for a signal (sent from the fuse operations) or a timeout to wake up (timeout because from fuse we cannot detect connectivity changes) and check if new entries were added to the journal. If the phone has Internet connection at the time, perform those operations and mark them as done. We use a hashmap to represents the journal, where the key of the stored values is an incremental id, and the value is a struct that contains information about an operation. We can use the contents of the journal to roll back any changes that lead to an inconsistent state, or to perform the remote operation. The struct used is described in Listing 4.4.

```
   typedef enum {
    flush,
    create,
    unlink
} operation;

   struct journal_entry {
   operation op;
   const char *path;
   time_t timestamp
};
```

Listing 4.4: Journal struct implementation

In this module we have implemented the following functions:

init_worker_thread Used to create and initialize a worker thread that locks waiting for new journal entries to be added in order to dispatch them;

add_journal_entry Used to add a new journal entry;

get_journal_entry Used to get the next journal entry (get the one that has the lower time stamp);

remove_journal_entry Used to remove an entry from the journal;

Unfortunately, this module is not working. The worker thread is killed by the kernel in the Android OS. The application was tested with this module turned on in Ubuntu Linux Distribution, and everything works fine, but in the Android OS it does not. That means that in Android our system only works with an Internet connection and the write operations are instantly synchronized with the remote storage.

4.9 Fuse Operations

In this section, we look into detail how we use the previously described modules and how they work together in order to meet the objectives. The implementation of this functions dictates how

the FS will behave when it receives a request.

4.9.1 getattr

This function is used to get the metadata of the files that must be saved to a struct $stbuf$ passed in the parameter list. If the metadata is not cached, we get the metadata of the file from $n-1$ CSP accounts and then add it to the cache. If we do not have at least $n-1$ chunks of the file (file present in $n-1$ out of the n CSP accounts), we return an error as the file is not available at the moment either because it was deleted or the CSP is down.

4.9.2 readdir

This function is used to list the contents of a folder, in order to simplify the implementation our system do not allow the user to create additional folders, so it will always list the root contents or return an error.

To perform this operation, if the contents of the folder is not in cache, we use the cloud abstraction module function $read\_dir$ to get the root contents of all the CSP accounts, then we filter only the ones that are present in at least $n-1$ accounts, that are the available files.

Before returning the directory contents, we add them to the cache using the function $add\_folder\_contents$ from the cache module.

4.9.3 create

This is the function called to create a new file in the FS. We first verify if the file does not already exist, otherwise we proceed to create it in the local cache, and save a file pointer in the struct $fuse\_file\_info$. This file handler will be used later in the $write$ and $flush$ functions. We do not immediately send the file to the clouds because when this function is called it is much likely to be followed by writes, and then by a flush (the flush is always called, even if there are no reads) and we update the remote space here.

We also create a $meta\_entry$ entry to save the metadata of the file in cache and also the location of the cached file (created in the beginning of the operation) in local storage.

4.9.4 write

This function writes data to an open file; we can directly access the file using the file handler previously set up in $open$ function. After the write, we must update the cache value for last access, because of the LRU cache and mark the file as inconsistent in cache (not remotely updated) as we are only updating it in the clouds when a $flush$ is called to reduce the number of networking operations. Note that in this function we should return exactly the number of bytes requested, except on error.

4.9.5 open

This is one of the most important functions as is there that we get the files from the clouds if they are not already in the local cache.

We start by checking if the file is already in the local cache, if it is we open the file and store its file handler in the $fuse\_file\_info\ fh$ field, the file handler is used to give access to the file in the subsequent operations (e.g. write or flush).

If the file is not in the local cache we check if the file is not larger than the total size of our cache. If it fits in cache we must check if it is available, where being available means that we have at least $n - 1$ providers available containing the chunk. After confirming that the file is available we proceed to download the necessary chunks using and checking their integrity. If a provider is down or the integrity check of a chunk fails, the get_file function automatically downloads the redundant chunk and calculates the missing one (using the availability module function to calculate the parity between the other chunks). Finally this function merges the chunks recreating the encrypted file. The get_file also removes the headers of the file parsing the IV to the variable iv as we need it for the next step, that is the decryption of the file using the $encrypt\_decrypt\_file$ function of the confidentiality module.

After obtaining the original file, we add it to the cache using the function add_file_cache and finally open it and add file handler the $fuse\_file\_info\ fh$ field.

4.9.6 read

The read function is only called after an open for the same file. That means that the file we want to read from is already in the local cache, so we just make a read from that file. In the $open$ function we have set up a file handler ($fi->fh$) to the target file, so we can use it to make the read. After the read we must update the cache value for last access, because of the LRU cache.

This function does not need to make any changes in the remote files.

4.9.7 flush

This function requests all characters to be written to the controlled sequence, meaning that we must make the writes to the file effective. This is where we update the remote storage.

We first make a flush to the local storage, then check if the file is not already updated in the remote storage (a flush can be called with no changes to a file); if it is not we call a function to do so ($update\_remote\_file$). Before returning we add the file the local cache if it is not already there.

The $update\_remote\_file$ function does the necessary operations to generate ensure file availability, integrity and confidentiality. It starts by generating a random IV, and encrypting the file using the IV and the key in $fs\_state->key$ with the confidentiality function $encrypt\_decrypt\_file$. Then we split the file in $n-1$ chunks, where n is the number of CSP accounts , and calculate the redundancy chunk using the functions of the availability module. After this we have a chunk per CSP account, we just calculate the HMAC of each chunk, using the integrity module. Finally, we add a header containing the HMAC and IV to the chunks and use the cloud abstraction module to dispatch the files to each CSP.

4.9.8 unlink

This function is used to delete a file, that is a simple process, we iterate over the configured CSP accounts and send a delete request for the respective chunk, then we remove the file from cache (to free space if it is in local cache), and finally we invalidate the meta-data cache of the file as we will not need it anymore.

4.10 Compiling for Android

After the implementation of the fuse operations, we need to generate a binary file that runs on the Android OS, for that we use the ndk-build script that launches the build scripts at the heart of the NDK, in order to generate binaries compatible with the Android OS.

The ndk-build script requires two makefiles in order to build our sources: the Android.mk file, which defines properties specific to individual modules, or libraries, and the Application.mk file, which defines properties for all the modules that we use.

In the Application.mk file, we have only defined the target architecture (`APP_ABI := armeabi-v7a`), we are compiling only for $armeabi - v7a$ that is the most used architecture for smartphones.

In the Android.mk file we first compile the dependencies as static libs, and then include them in the main module. This process was complex because the Android OS was missing a lot of core libs that we needed, so we had to get the source code and generate mkfiles to build them as static libs. We then build two executable files, the mcfs module that is our FS and fusermount module that is used to unmount the file system.

After having these two files correctly written, to build the binary files we just need to be in the root folder of the project and call the command $ndk - build$. As a result of this process we have two binary files (fusermount and mcfs), now we can send them to a rooted Android phone and execute mcfs with the necessary parameters (e.g. $./mcfs\ mount\_dir\ 1024\ crypt\_pass\ hmac\_pass$) to mount the FS and use $fusermount\ -u\ mount\_dir$ to unmount it.

4.11 Android Application

The Android application provides a visual interface that allows users to manage the system. The main features are: generating the configuration file with the access tokens to the CSPs, mount the FS and unmount it.

In the figure 4.2 we can see the main menu layout where we can set the passwords, the cache size, the mount point and mount or unmount the FS.

The CSP account configuration menu, see figure 4.3, allows the users to add or remove CSP

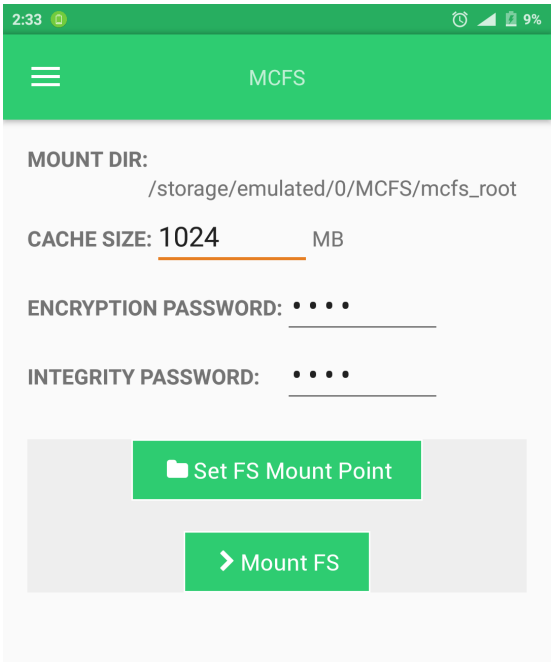


Figure 4.2: The main menu of our Android application, where we can set the passwords, mount and unmount the FS.

accounts to the configuration file.

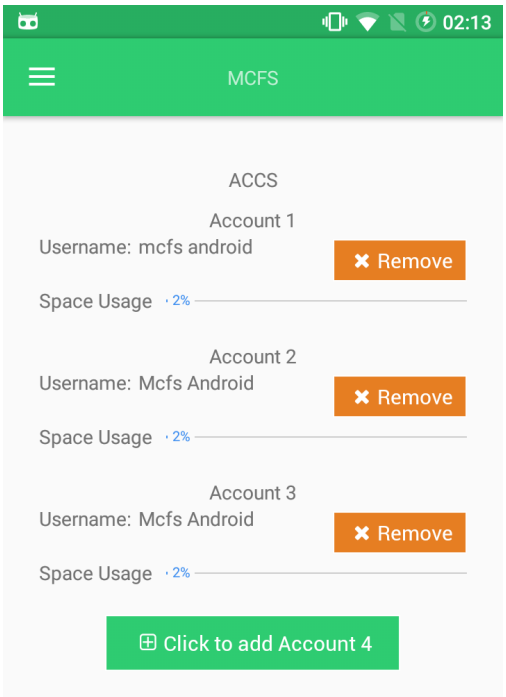


Figure 4.3: The menu of our Android application where we can set manage the CSP accounts.

To add an account, we use the OAuth authorization framework (see Figure 4.4) that generates

the access tokens to be later used in the FS implementation.

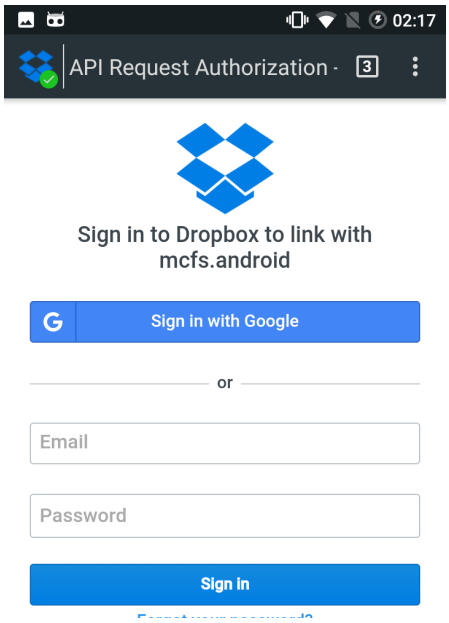


Figure 4.4: To add an CSP account to the system we use the OAuth authorization framework.

The configuration file is a json with an array ($accs\_ids$) with the ids of the accounts and another array containing the authentication tokens for each account ($access\_token$, $token\_key$ and a $token\_secret$) 4.5.

```
{"accs_ids":[0,1,2],
   "0":{
      "access_token":"ilL8Daq4ojAAAAAAAAAAESU6g6ATWS1JK",
      "token_key":"6aso0c2buznapjcc",
      "token_secret":"3advtmu5ga6y6on"
   },"1":{
      "access_token":"ilL8Daq4ojAAAAAAAAAAESU6g6ATWS1JK",
      "token_key":"f1a8cyggalblp3gk",
      "token_secret":"0000000000000000"
   },"2":{
      "access_token":"ZQCtpXAYMC5n00yoy2npyZd3M8IuWCNAd",
      "token_key":"rj3pwqyx5avbfbp0",
      "token_secret":"ig19ae3gtppa9ib"
   }
}
```

Listing 4.5: Configuration file with the authentication information to the clouds

After mounting the system, it will stay mounted until the user explicit hits the unmount button. The folder chosen as the mount point will list the FS files and allow to access them like they were local files also allowing to add new files to the system just by copying them into the folder.

Chapter 5

Testing and performance analysis

In this chapter, we test, validate, and evaluate the implemented system. First, we describe the test setup, and then we explain how to use the application, presenting some preliminary results. Then we make a performance evaluation and comparison to other solutions in order to validate the viability of the solution. Finally we present the preliminary results of the implementation (the FS and the Android application), showing the steps we need to configure the system and mount the FS.

5.1 Application configuration and preliminary results

In this section we describe the test setup. All the tests here presented were made using a Samsung Galaxy S3 running a rooted CyanogenMod Android OS on its 6.01 version (as we can see on figure 5.1), connected the same wireless network with a contracted speed of 50Mbps.

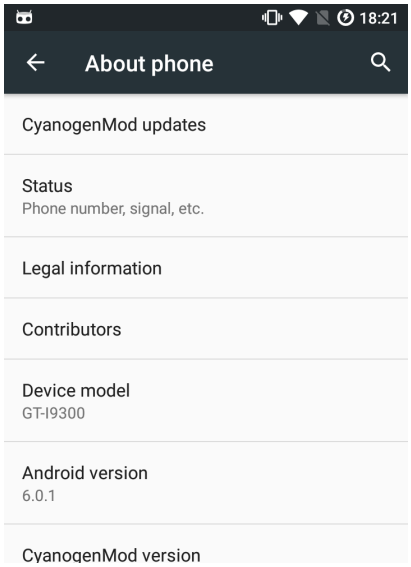


Figure 5.1: The setting menu of the phone used to do the tests

First, we install the Android application that also contains the binary file of the file system generated with ndk-build. When we open the application the first thing we need to do is to add at least three CSP accounts. For that, we have an account configuration menu, as shown in Figure 5.2. To add an account we must press the $AddAddcount$ button that initiates an authorization request.

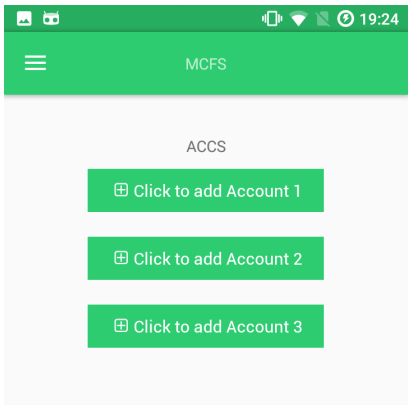


Figure 5.2: Android account configuration menu as open for the first time

In Figure 5.3, we show the menu used to login in the Dropbox CSP.

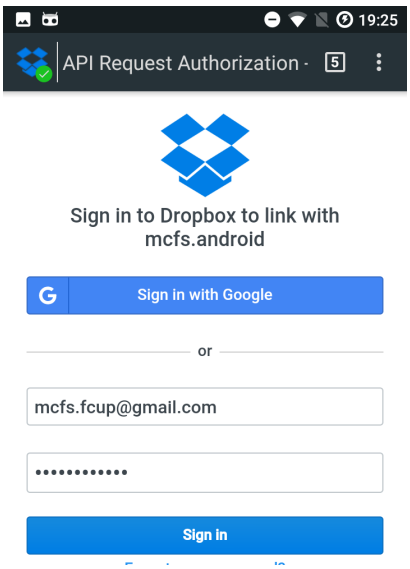


Figure 5.3: Login into Dropbox CSP to add an account to the system

After logging in, we need to allow the access to the API of the CSP, in order to generate the access tokens that we will need to communicate with the account. In figure 5.4 we can see the page shown to authorize the API access for the Dropbox CSP.

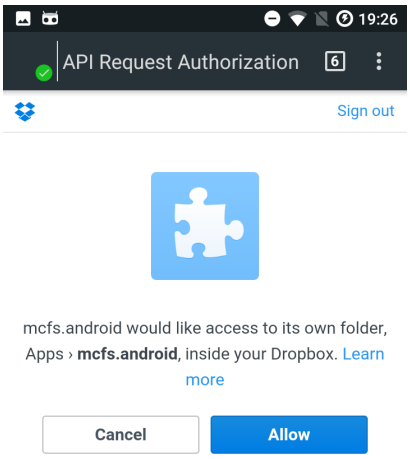


Figure 5.4: After loggign in we need to allow the the API access

After allowing the API access, the account will be added to the system and the API access tokens saved in a configuration file. In figure 5.5 we can see that now the account is listed in the configurated accounts. In this menu we can also track the used space in each account.

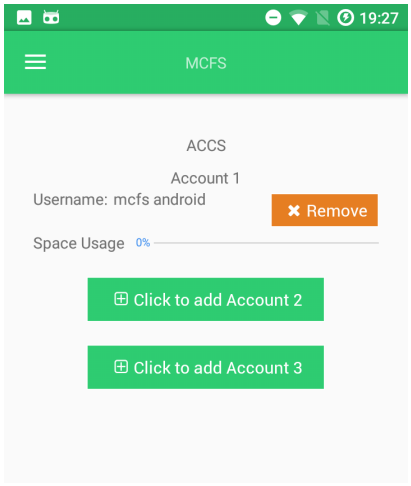


Figure 5.5: Accounts configuration menu with one account configurated

We need to add at least two more accounts in order to be able to use the system. The procedure is the same as for the first account. After adding the necessary accounts, we can now configure other parameters of the system. We need to set the total size of the local cache, the mount point and the integrity and confidentiality passwords as we can see in figure 5.6.

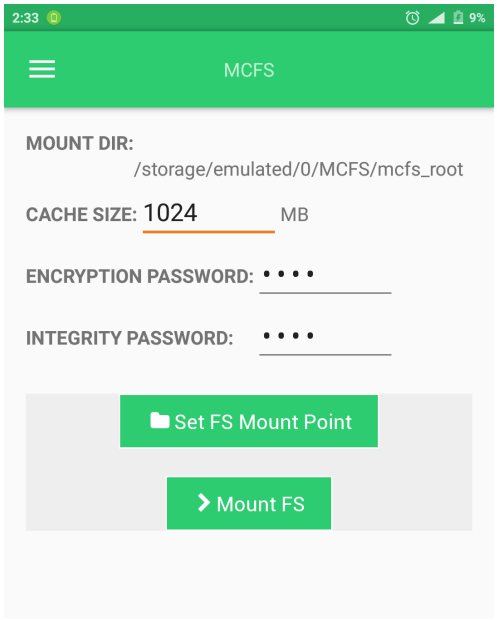


Figure 5.6: The configuration menu that allows to mount the system

After configuring the above-described parameters, the user can now press the $MountFS$ button. In figure 5.7 we can see the chosen mount point folder contents before we mount the system,

using a regular file explorer.

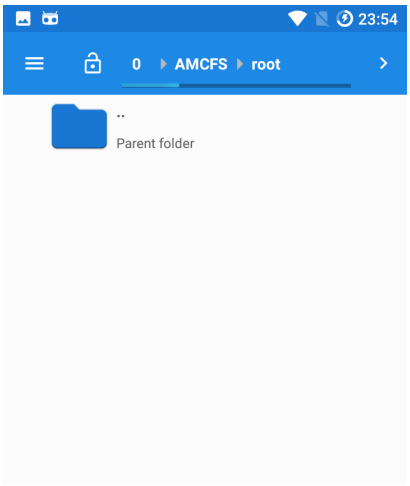


Figure 5.7: Mount point folder contents before mount

After pressing the $MountFS$ button, we copy the configuration file to the parent directory (the $mcfs$ binary is expecting it to be there) of the mount point and run the $mcfs$ binary file with the configured parameters. Moreover, we change the $MountFS$ button to a button that allows the user to unmount the system as we can see in figure 5.8.

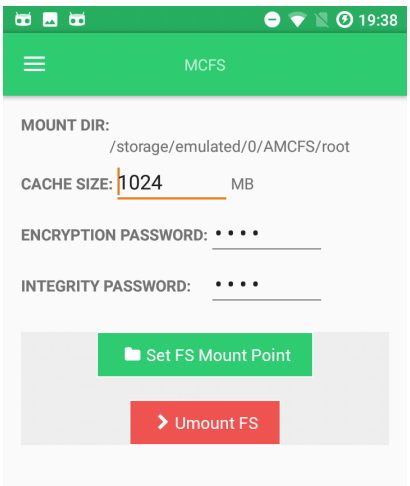


Figure 5.8: The configuration menu that allows to unmount the system

Finally, we can use the folder as a regular local folder, in order to get and store files in our FS. In figure 5.9 we can see that now the folder chosen as mount point lists the files stored in the FS.

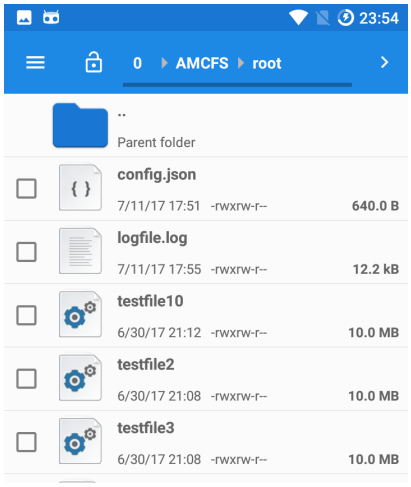


Figure 5.9: Mount point folder contents after mount

The storage overhead for each file, assuming that we have configured three CSP accounts, is 33% which means that if we have 6Gb of total space, 2Gb per account we can store up to 4Gb. This overhead goes down as the number of CSP accounts goes up.

Using the system we noted that some applications do not work properly when operating with files of our FS. This is due to the time that the operations take, that is higher than the operations in the local storage. The cause of that issue is the low timeout values in the applications, as they are not designed to work with network file systems. This happened mostly when the applications were trying to open a file. The write operations behaved normally almost every time.

We made tests simulating the unavailability of one CSP, and the system successfully detected and behaved correctly, getting the redundant chunk and calculating the missing one in order to present the correct file to the user.

We also tested a case where the integrity of file was violated; the system detected and successfully recovered from that by getting the redundant chunk and calculating and replacing the one that had his integrity violated.

To test the cache system, we defined a cache size of 1Gb, and we stored files with various file sizes summing up 2Gb. Then we opened some of them and the system correctly replaced in the cache the ones that were not accessed for longer.

5.2 Performance Analysis

In this section we present some performance test that we consider relevant to better evaluate the behavior of our solution. We previously explained the storage overhead in Subsection 2.2; now we want to understand the performance overhead.

The fact that we are using FUSE for the implementation implies some overhead that we can not measure in code, because of that the time measurements were made on the command line using the command line tool $time$, that measures the time that a command takes to run. Doing, for example, $time\ cp\ file1\ root/$, where root is the mount point of our system will measure the time that our system takes to store a file.

We measured the time that our system takes to access and store a file, separating the time that it takes to access a file in two categories: with and without the file in cache. For the access operation we also measured the time when we have a CSP unavailable. In all the tests, the measurements were repeated multiple times with different file sizes, being the average time calculated by repeating the operations twenty times. The results are presented with plots where the vertical blue bar represents the average duration of an operation, and the black vertical bar represents the standard deviation.

The results of the first test correspond to the store operation, where we add a file to the FS and measure the time that it takes. Analyzing the Figure 5.10 we can see that as expected the average time to store a file increases with the file size. The growth of the average time is not proportional to the file size: from the 1Mb file to the 10Mb file we have an increase of about 1.7 times; from the 10Mb file to the 100Mb file the growth is of about 5.3 times. From that, we can conclude that as the file size increases the overhead of our operations (to ensure availability, integrity, and confidentiality) in the file also grows, because if it were only for the network time the growth would be more proportional to the file size.

Then we tested the average time to retrieve a file from the file system in the different scenarios that we can face. The results are shown in Figure 5.11. We can see the huge difference between having and not having the files in the cache. This is due to the fact that the files in cache are stored in clear text (not encrypted); we do not need to get them from the network, and, moreover, we do not need to make any operations on the file. We can also see that when we have one CSP

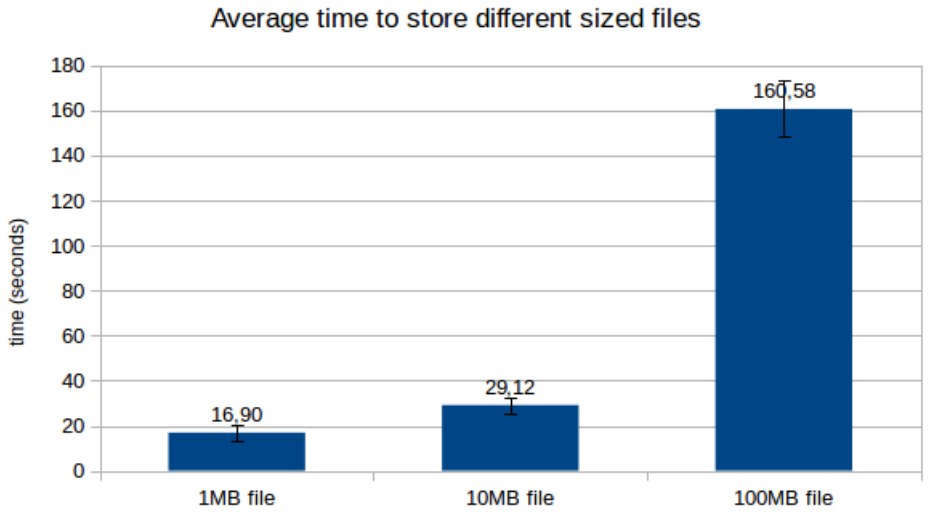


Figure 5.10: The average time to store different sized files

down and the file is not cached we have an overhead. This is due to the fact that we need to perform one extra step to recalculate the missing chunk.

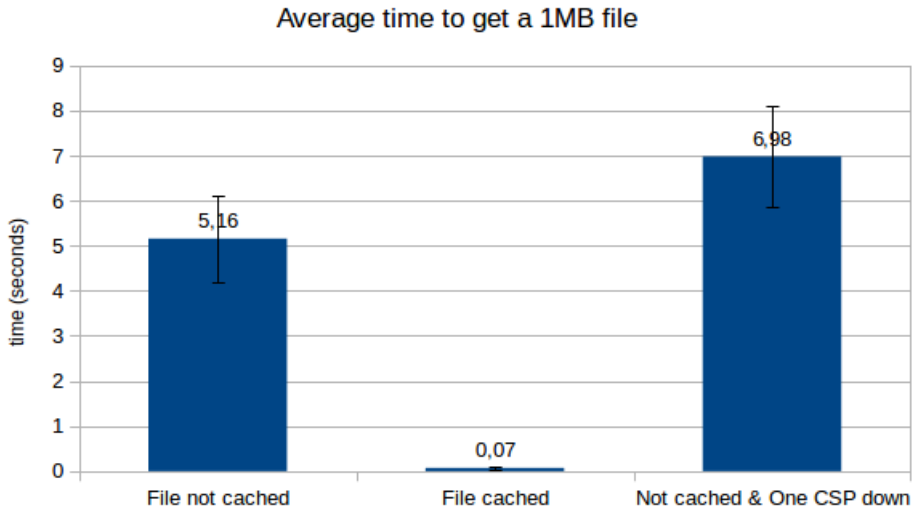


Figure 5.11: The average time to get a 1Mb file stored in our system, in different situations

We repeated the previous test only changing the file size. The results can be seen in Figure 5.12 and the conclusions that we can draw from them are basically the same.

Again we repeated the previous test changing file size, and the results can be consulted in Figure 5.13, the conclusions of this trial are basically the same as the two previous ones, but now we can make some comparisons with the other two. We can note that the average time to get a

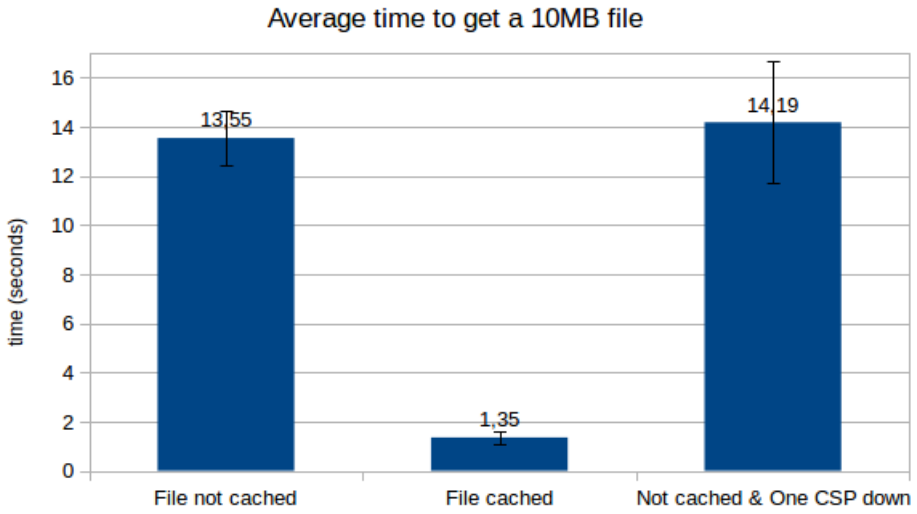


Figure 5.12: The average time to get a 10Mb file stored in our system, in different situations

file not cached is not proportional the file size, being the increase from the 1Mb file to the 10Mb file of about 2.6 times and from the 10Mb file to the 100Mb file of about 5.4 times. Similarly to the store operations, this is due to the overhead of the operations over the file. As the file size grows, so does the time that it takes to encrypt it, calculate the HMAC and the other operations described in chapter 3. The average time that takes to get a cached file is almost proportional to the file size. This is because the only performed operation is a system $open$, and its duration increases proportionally to the file size.

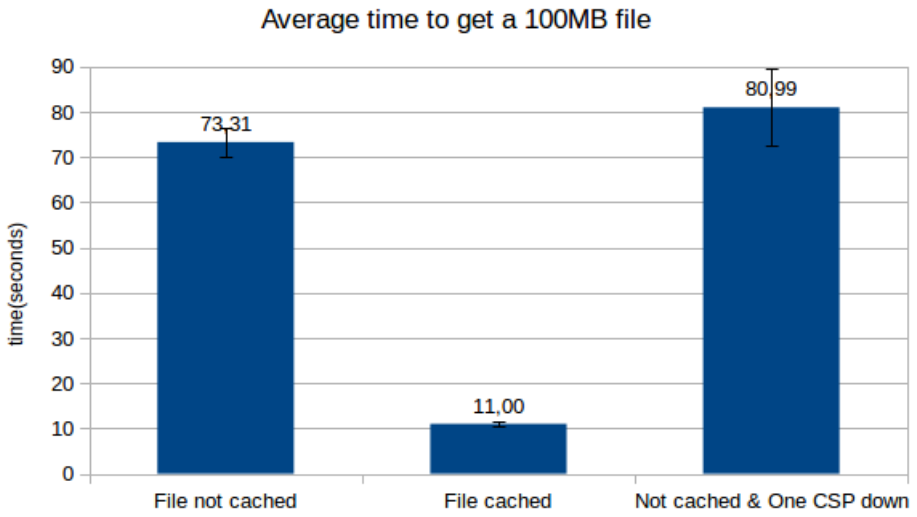


Figure 5.13: The average time to get a 100Mb file stored in our system, in different situations

We also compared our system with a conventional single cloud application (Android Dropbox application). As we can see in Figure 5.14, the average time it takes to store a file with our system is about 2.6 times higher compared to Dropbox Android application. The average time to get a previously stored file in our system is about 2.1 times higher than with Dropbox Android application, as we can see in Figure 5.15.

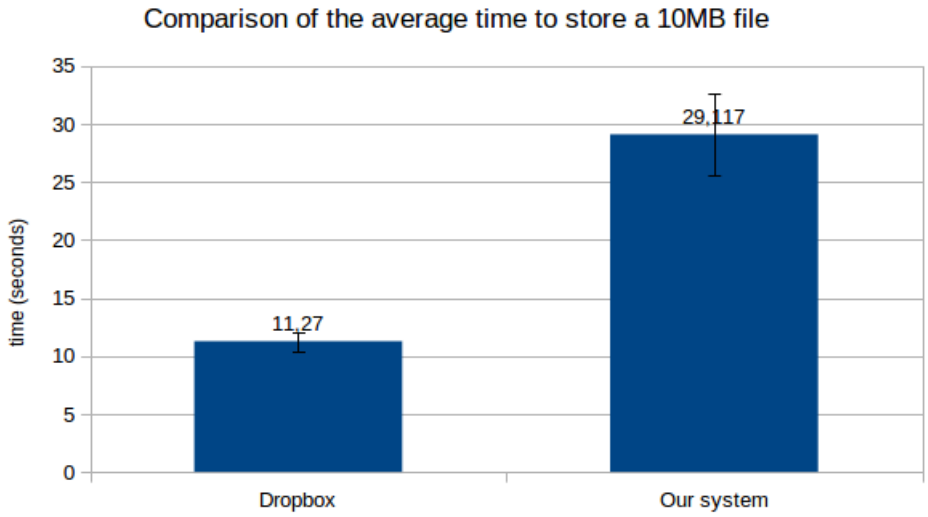


Figure 5.14: The average time to store a 10Mb file with our system and Dropbox Android application

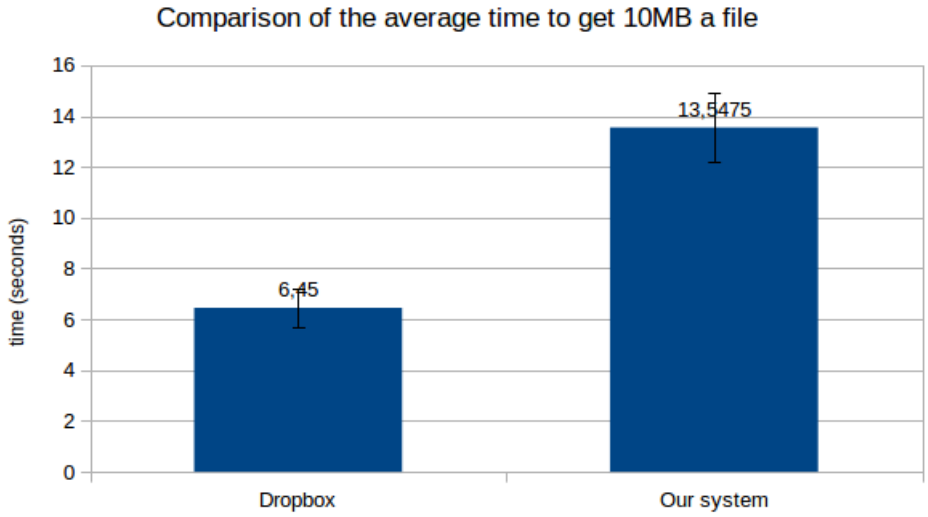


Figure 5.15: The average time to get a 10Mb file with our system and Dropbox Android application

The less favorable performance of the get and store operations in our system are due to the fact that it needs to perform more computations for the file operations (to add availability, integrity, and confidentiality) and it needs to get file chunks from different sources, while Dropbox

application gets the file from one source and do a lot less computation over the file.

The measurements of the times of the Dropbox application get and store operations were made using the Android App, measuring the time from the click on the button (send file/download file) to the end of the task.

The collected data helped us understand the impact of the cache module in the system performance, providing a great performance gain after the first access to a certain file. In general the performance (access times) is very acceptable, taking into account that that the system provides confidentiality, integrity and improved availability of the files.

Chapter 6

Conclusions and Future Work

We successfully implemented an Android FS that has backing storage public CSPs and is able to detect and recover from integrity violation, recover from one CSP failure by adding redundancy using coding techniques, and ensure file confidentiality using cryptographic techniques.

The availability of the files is improved using a coding technique that generates redundant information. The integrity and confidentiality of the stored files is achieved using cryptographic techniques.

Users configure the file system adding CSP accounts and setting passwords to be utilized in the cryptographic operations. Then they can mount the FS in a local folder, and use it as it was local storage. The fact that the files are remotely stored is completely transparent to the user, except the performance because of the network nature of the operations.

The cipher used to achieve confidentiality of the files is AES-256, that is the industry standard. It is considered secure for the coming years as there is no evidence of successful attacks to it. Each CSP only contains a chunk of the encrypted file. An attacker that manages to get access to a CSP account can only see encrypted chunks of the original files, thus being unable to retrieve any information about the original content.

To give the system the ability to detect integrity violations and recover from them we used HMAC, using the SHA 2 algorithm with a 256 key size (SHA/256). Therefore, we can detect if a previously stored chunk of a file has been changed since it was stored.

The availability of the stored files is improved using a coding technique that generates a redundant chunk stored in an extra CSP, giving the system the ability to tolerate the failure of one CSP.

The performance degradation and storage overhead, when compared to other solutions, is the price we have to pay for the security and data availability of our solution. The overhead does not compromise any of the user direct operations and it is acceptable taking into account the security and availability provided by the system. However, some applications do not tolerate the time overhead of the storage operations, due to its low timeout values.

6.1 Future Work

Although the objectives of this work are met there are some aspects that can be improved. For future work we consider the following improvements:

Cache Replacement Policy In this work we only implemented one cache replacement policy (LRU). The implementation and testing other policies can reveal some interesting facts about the access times. The system can even let the user choose what cache replacement policy to use;

Journaling Module The journaling module is implemented but not working in the Android OS. Due to the lack of time we could not find the origin of this error. This issue needs to be addressed in the future work. Also, we can improve the journaling module to better recover from eventual inconsistency states and;

Availability For most of the cases, the implemented method to improve the availability of stored files is sufficient. However, we can improve this even further allowing the system to resist to the simultaneous unavailability of more than one CSP;

Cloud Access Module In this module, due the lack of time and tools, we only implemented the access to the DropBox CSP. It is important to implement the access to other CSPs in order to get the system fully functional;

Recover File System Create a module to allow the users to replace a configured CSP account and automatically migrate the data to the new account (generate the correspondent file

chunks from the other CSP accounts). This feature is necessary to replace a CSP account of CSP that goes off business and maintain the system fully functional.

References

[1] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[2] Iulia Ion, Niharika Sachdeva, Ponnurangam Kumaraguru, and Srdjan Čapkun. Home is safer than the cloud!: privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security*, page 13. ACM, 2011.

[3] Wenjin Hu, Tao Yang, and Jeanna N Matthews. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review*, 44(3):110–115, 2010.

[4] Brandon Butler. And the cloud provider with the best uptime in 2015 is.. Online, November 2016. Avaliable at http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html, Accessed December 05, 2016.

[5] Christophe Cérin, Camille Coti, Pierre Delort, Felipe Diaz, Maurice Gagnaire, Quentin Gaumer, Nicolas Guillaume, J Lous, Stephane Lubiarz, J Raffaelli, et al. Downtime statistics of current cloud solutions. *International Working Group on Cloud Computing Resiliency, Tech. Rep*, 2013.

[6] Erica Naone. Technical problems at ma.gnolia.com raise questions about how social media is being protected. Online, November 2009. Avaliable at https://www.technologyreview.com/s/412041/are-we-safeguarding-social-data, Accessed December 05, 2016.

[7] Samuel Gibbs. Dropbox hack leads to leaking of 68m user passwords on the internet. Online, August 2016. Avaliable at https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach, Accessed December 05, 2016.

[8] Rich McCormick. Hack leaks hundreds of nude celebrity photos. Online, November 2014.

Avaliable at http://www.theverge.com/2014/9/1/6092089/nude-celebrity-hack, Accessed December 05, 2016.

[9] Justice Opara-Martins, Reza Sahandi, and Feng Tian. Critical review of vendor lock-in and its impact on adoption of cloud computing. In *Information Society (i-Society), 2014 International Conference on*, pages 92–97. IEEE, 2014.

[10] Dave Chaffey. Mobile marketing statistics compilation. Online, November 2016. Avaliable at http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics, Accessed December 05, 2016.

[11] IDC. Smartphone os market share, 2016 q2. Online, November 2016. Avaliable at https://www.idc.com/prodserv/smartphone-os-market-share.jsp, Accessed December 05, 2016.

[12] IEEE SA. Ieee standard for information technology - portable operating system interface (posix(r)). Online, August 2008. Avaliable at http://standards.ieee.org/findstds/standard/1003.1-2008.html, Accessed December 05, 2016.

[13] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. *INC, IMS and IDC*, pages 44–51, 2009.

[14] Shahriar Hossain. Cloud delivery model (saas vs paas vs iaas). Online, October 2014. Avaliable at https://learnwithshahriar.wordpress.com/2014/10/16/cloud-delivery-model-saas-vs-paas-vs-iaas/, Accessed December 17, 2016.

[15] Leoinspace. A high level architecture of cloud storage. Online, May 2014. Avaliable at https://commons.wikimedia.org/wiki/File:Cloud_storage_architecture.png, Accessed December 17, 2016.

[16] Gillware. Raid-1 recovery services. Online, September 2015. Avaliable at https://www.gillware.com/raid-1-recovery-services/, Accessed December 05, 2016.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop

distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[19] Ulf Troppens, Rainer Erkens, Wolfgang Muller-Friedt, Rainer Wolafka, and Nils Haustein. *Storage networks explained: basics and application of fibre channel SAN, NAS, iSCSI, infiniband and FCoE*. John Wiley & Sons, 2011.

[20] Margaret Rouse. Raid 4 (redundant array of independent disks). Online, December 2014. Avaliable at http://searchstorage.techtarget.com/definition/RAID-4-redundant-array-of-independent-disks, Accessed December 17, 2016.

[21] Corentin Debains, Pedro Manuel Alvarez-tabio Togores, and Ioan Raicu. Evaluating information dispersal algorithms. In *1st Greater Chicago Area System Research Workshop*, 2012.

[22] Mingqiang Li. On the confidentiality of information dispersal algorithms and their erasure codes. *arXiv preprint arXiv:1206.4123*, 2012.

[23] Michael O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.

[24] Franco P Preparata. Holographic dispersal and recovery of information. *IEEE Transactions on Information Theory*, 35(5):1123–1124, 1989.

[25] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[26] Alexandre Soro and Jérôme Lacan. Fnt-based reed-solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5. IEEE, 2010.

[27] Arto Salomaa. *Public-key cryptography*. Springer Science & Business Media, 2013.

[28] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[29] Hugo Krawczyk. Secret sharing made short. In *Annual International Cryptology Conference*, pages 136–146. Springer, 1993.

[30] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[31] TOPDOX Documents. Topdox ficheiros & cloud docs. Online, May 2017. Avaliable at https://play.google.com/store/apps/details?id=topresearch.topdocs, Accessed September 05, 2017.

[32] bobrofon. Easysshfs. Online, May 2017. Avaliable at https://play.google.com/store/apps/details?id=ru.nsu.bobrofon.easysshfs, Accessed September 05, 2017.

[33] INESC TEC. Safecloud photos. Online, May 2017. Avaliable at hhttp://www.safecloud-project.eu/consortium/inesctec, Accessed September 07, 2017.

[34] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[35] Ricardo Mendes, Tiago Oliveira, Alysson Bessani, and Marcelo Pasin. C2fs: um sistema de ficheiros seguro e fiável para cloud-of-clouds. *INForum12, September*, 2012.

[36] Shadi R Masadeh, Shadi Aljawarneh, Nedal Turab, and Aymen M Abuerrub. A comparison of data encryption algorithms with the proposed algorithm: Wireless security. In *Networked Computing and Advanced Information Management (NCM), 2010 Sixth International Conference on*, pages 341–345. IEEE, 2010.

[37] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for preimages: attacks on skein-512 and the sha-2 family. In *Fast Software Encryption*, pages 244–263. Springer, 2012.

[38] Darryl L Willick, Derek L Eager, and Richard B Bunt. Disk cache replacement policies for network fileservers. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 2–11. IEEE, 1993.

[39] William Krier and Erik Liska. Fuse design document. Online, May 2009. Avaliable at http://www.youblisher.com/p/31627-fuse/, Accessed September 06, 2017.

[40] Sven. Fuse structure. Online, May 2014. Avaliable at https://en.wikipedia.org/wiki/Filesystem_in_Userspace#/media/File:FUSE_structure.svg, Accessed September 06, 2017.